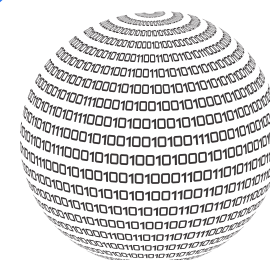
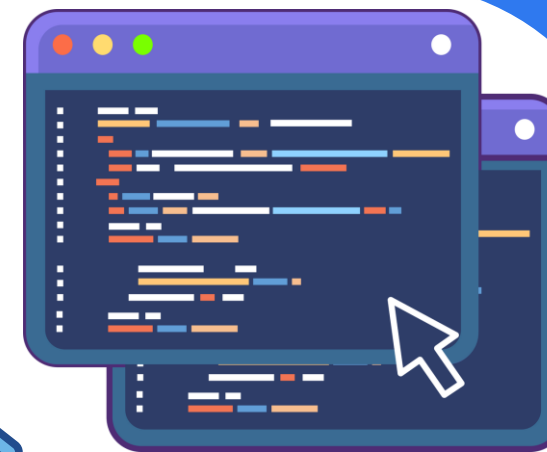
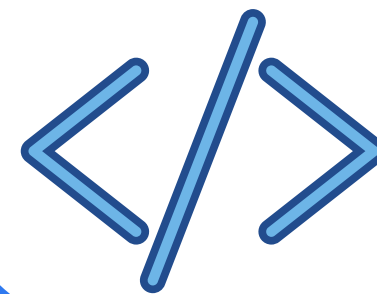




BAN HỌC TẬP
HỆ THỐNG THÔNG TIN

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

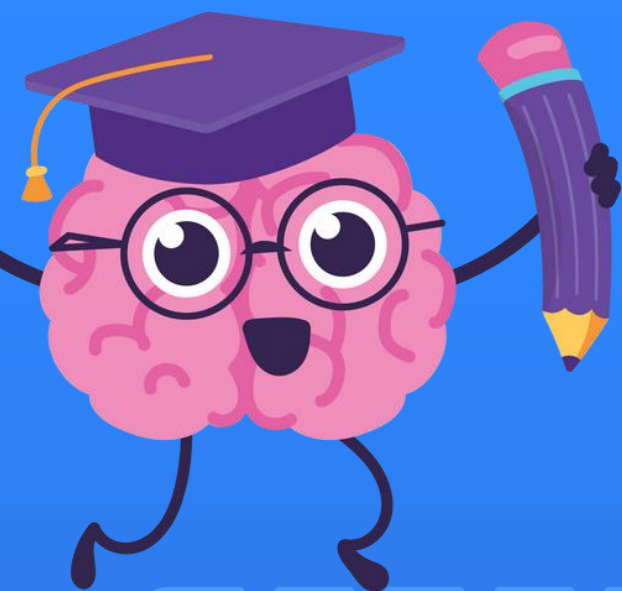
- Trần Thị Kiều Trâm
- Dương Trọng Toàn





CHƯƠNG II TỔNG QUAN VỀ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG





CHƯƠNG II

TỔNG QUAN VỀ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

1. Các phương pháp lập trình
2. Một số khái niệm cơ bản trong OOP
3. Các đặc điểm quan trọng của OOP
4. Phân tích, thiết kế và lập trình hướng đối tượng



BAN HỌC TẬP
HỆ THỐNG THÔNG TIN

4



CÁC PHƯƠNG PHÁP LẬP TRÌNH



LẬP TRÌNH KHÔNG CÓ CẤU TRÚC

- Là phương pháp xuất hiện đầu tiên. Phương pháp này đơn giản chỉ là viết tất cả mã lệnh vào 1 hàm main duy nhất và chạy.
- Ngôn ngữ sử dụng phương pháp này là Assembly (hợp ngữ).
- Nhược điểm của phương pháp này:
 - Chỉ sử dụng biến toàn cục dẫn đến rất tốn bộ nhớ.
 - Vì có những đoạn chương trình cần sử dụng lại nhiều lần nên dẫn đến lạm dụng lệnh goto.
 - Khó hiểu, khó bảo trì, không thể tái sử dụng.
 - Khó phát triển các ứng dụng lớn.

VÍ DỤ

```
10      k =1
20      gosub 100
30      If y > 120 goto 60
40      k = k+1
50      goto 20
60      print k, y
70      stop
100      y = 3*k*k + 7*k-3
110     return
```




LẬP TRÌNH CÓ CẤU TRÚC

- Chia chương trình lớn ra thành các chức năng, mỗi chức năng được đưa vào 1 hàm. Khi cần dùng đến chức năng nào thì ta sẽ gọi hàm tương ứng.
- Mỗi chương trình con lại có thể chia nhỏ ra nữa.
- Hầu hết các ngôn ngữ lập trình đều hỗ trợ phương pháp này.

ƯU ĐIỂM, NHƯỢC ĐIỂM

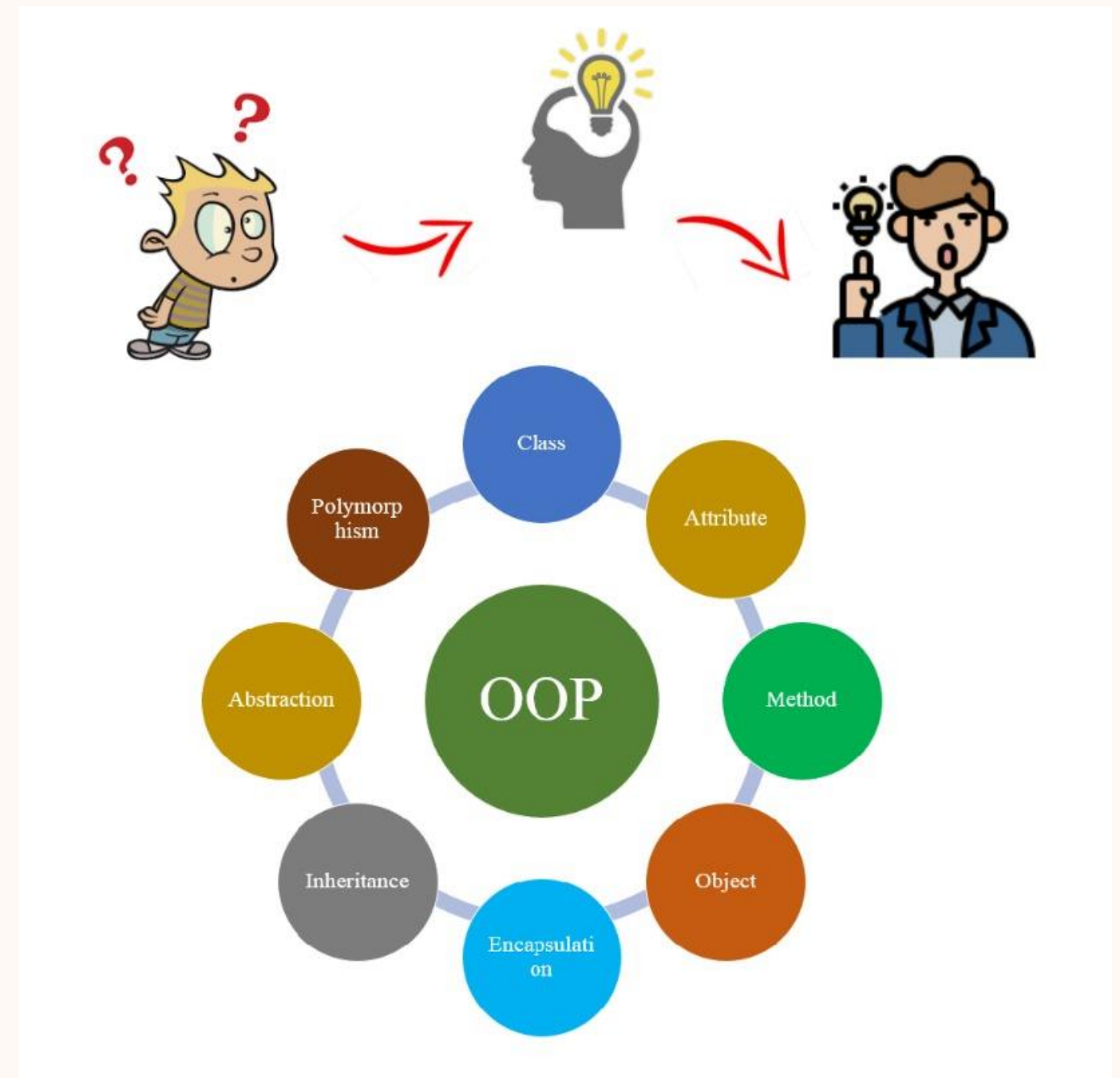
- Ưu điểm:
 - Chương trình được module hoá, dễ hiểu, dễ bảo trì.
 - Dễ dàng tạo ra các thư viện phần mềm.
- Nhược điểm:
 - Dữ liệu và xử lý tách rời.
 - Khi cấu trúc dữ liệu thay đổi sẽ dẫn đến thuật toán bị thay đổi.
 - Không tự động khởi tạo, giải phóng dữ liệu động.
 - Không mô tả được đầy đủ, trung thực hệ thống trong thực tế.



LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

- Với mong muốn xây dựng một phương pháp lập trình trực quan, mô tả trung thực hệ thống trong thực tế vì thế phương pháp lập trình hướng đối tượng ra đời.
- Lập trình hướng đối tượng là phương pháp lập trình lấy đối tượng làm nền tảng để xây dựng chương trình.
- Một định nghĩa khác về lập trình hướng đối tượng đó là phương pháp lập trình dựa trên kiến trúc lớp (class) và đối tượng (object).

MINH HỌA





MỘT SỐ KHÁI NIỆM CƠ BẢN TRONG OOP



ĐỐI TƯỢNG

- Trong lập trình hướng đối tượng, đối tượng được hiểu như là 1 thực thể: người, vật hoặc 1 bảng dữ liệu, ...
- Một đối tượng bao gồm 2 thông tin: thuộc tính và phương thức.
- Thuộc tính chính là những thông tin, đặc điểm của đối tượng.

Ví dụ: một người sẽ có họ tên, ngày sinh, màu da, kiểu tóc, ...

- Phương thức là những thao tác, hành động mà đối tượng đó có thể thực hiện.

Ví dụ: một người sẽ có thể thực hiện hành động nói, đi, ăn, uống, ...

LỚP

- Các đối tượng có các đặc tính tương tự nhau được gom chung thành lớp đối tượng.
- Một lớp đối tượng đặc trưng bằng các thuộc tính và các hành động (hành vi, thao tác) của các đối tượng thuộc lớp.

- Thuộc tính (Attribute): Một thành phần của đối tượng, có giá trị nhất định cho mỗi đối tượng tại mỗi thời điểm trong hệ thống.

- Thao tác (Operation): Thể hiện hành vi của một đối tượng tác động qua lại với các đối tượng khác hoặc với chính nó.

- Một đối tượng cụ thể thuộc một lớp được gọi là một thể hiện (instance) của lớp đó.



PHÂN BIỆT ĐỐI TƯỢNG VÀ LỚP

Lớp	Đối Tượng
Là một template chung cho tất cả các đối tượng, là một mô tả trừu tượng	Là một thể hiện của lớp
Chỉ được khai báo một lần	Có thể có nhiều đối tượng thuộc lớp
Khi khai báo thì không được cấp phát vùng nhớ	Khi khai báo thì được cấp phát vùng nhớ
Là một nhóm đối tượng giống nhau	Là những đối tượng cụ thể có thật



BAN HỌC TẬP
HỆ THỐNG THÔNG TIN

11

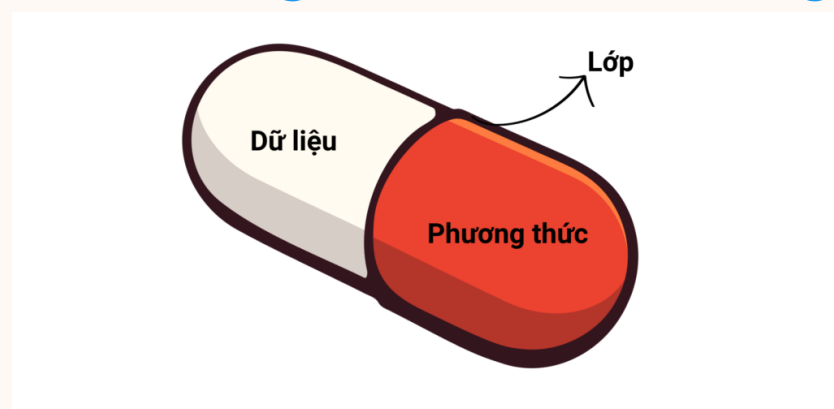


CÁC ĐẶC ĐIỂM QUAN TRỌNG CỦA OOP



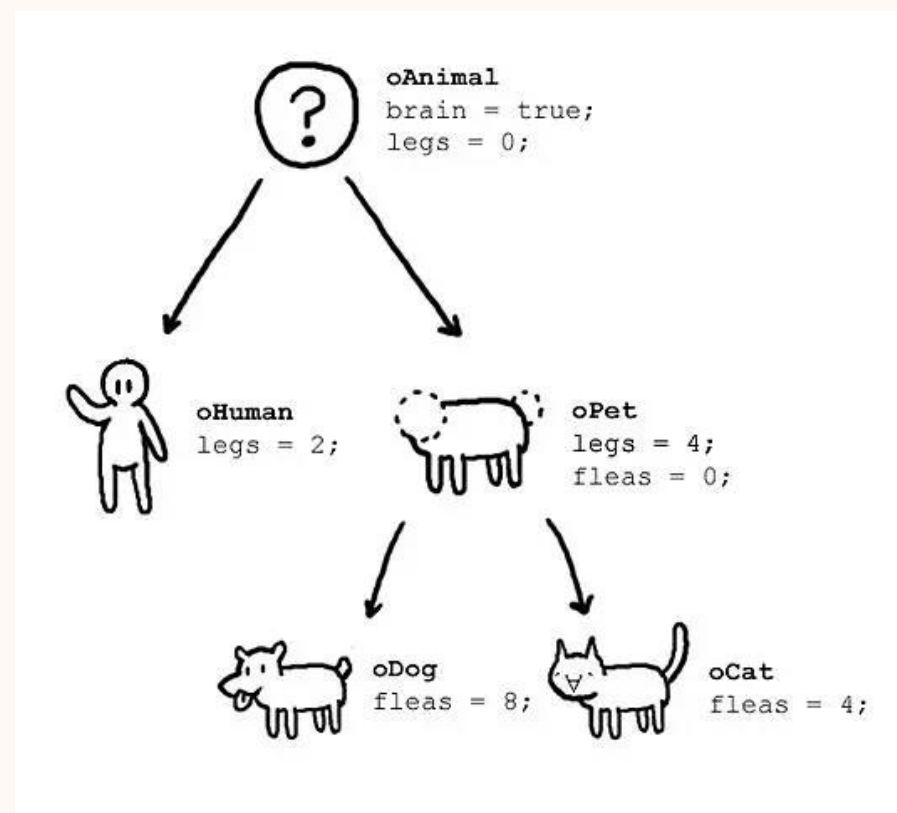
TÍNH ĐÓNG GÓI

- Các dữ liệu và phương thức có liên quan với nhau được đóng gói thành các lớp để tiện cho việc quản lý và sử dụng.
- Các hàm/ thủ tục đóng gói các câu lệnh.
- Các đối tượng đóng gói dữ liệu của chúng và các thủ tục có liên quan.
- Ngoài ra, đóng gói còn để che giấu một số thông tin và chi tiết cài đặt nội bộ để bên ngoài không thể nhìn thấy.



TÍNH TRỪU TƯỢNG

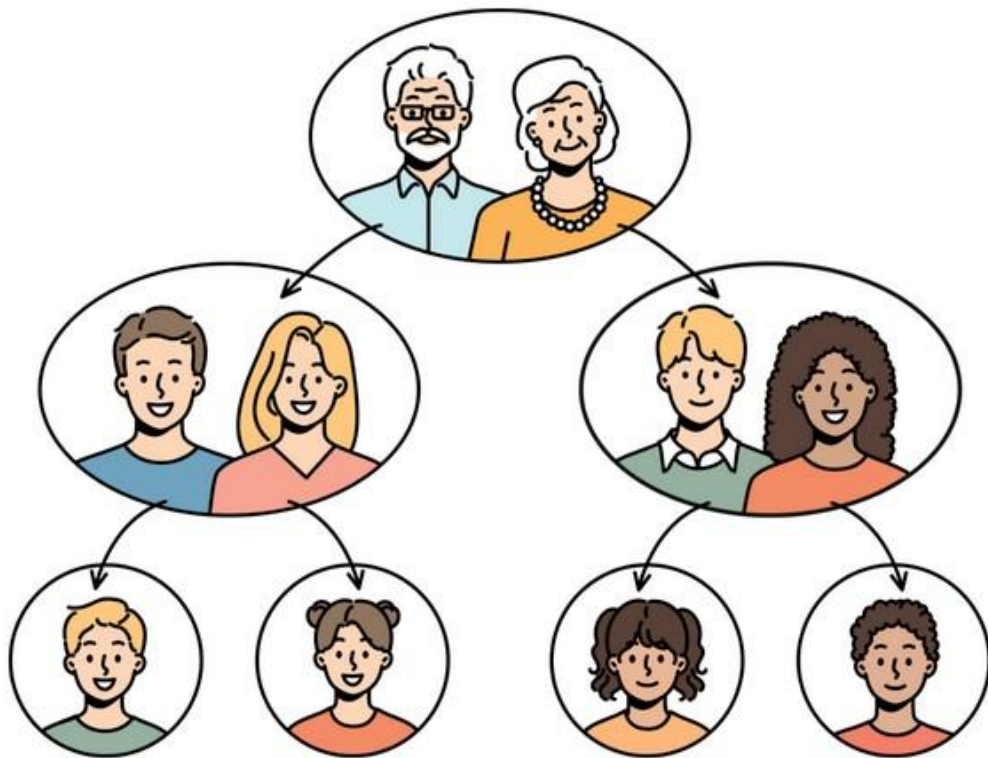
- Khi viết chương trình theo phong cách hướng đối tượng, việc thiết kế các đối tượng ta cần rút tỉa ra những đặc trưng chung của chúng rồi trừu tượng thành các interface và thiết kế xem chúng sẽ tương tác với nhau như thế nào.





TÍNH KÊ THỪA

- Lớp cha có thể chia sẻ dữ liệu và phương thức cho các lớp con, các lớp con khỏi phải định nghĩa lại, giúp chương trình ngắn gọn.



TÍNH ĐA HÌNH

- Là cơ chế cho phép tên một thao tác hoặc thuộc tính có thể được định nghĩa tại nhiều lớp và có thể có những cài đặt khác nhau tại các lớp đó.





Phân tích, thiết kế và lập trình hướng đối tượng



PHÂN TÍCH ĐỐI TƯỢNG

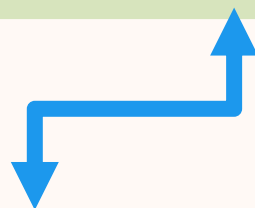
B1: Định nghĩa bài toán



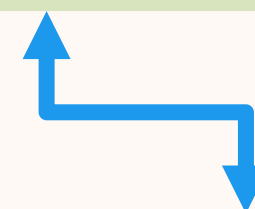
B2: Xây dựng các đặc tả yêu cầu của người sử dụng và của hệ thống



B3: Xác định các đối tượng và các thuộc tính của chúng



B4: Xác định hành vi
của các đối tượng



B5: Xác định mối quan
hệ giữa các đối tượng



THIẾT KẾ ĐỐI TƯỢNG (BOTTOM – UP)

- B1: Xác định các đối tượng trong không gian lời giải
- B2: Xây dựng các đặc tả cho các đối tượng, các lớp và mối quan hệ giữa chúng. quan hệ kế thừa, quan hệ thành phần, quan hệ về sử dụng.
- B3: Xây dựng cấu trúc phân cấp các lớp.
 - Theo nguyên lý tổng quát hóa.
 - Theo nguyên tắc sử dụng lại tối đa các thuộc tính và hàm của những lớp đã được thiết kế trước.
- B4: Thiết kế các lớp: bổ sung thêm những thuộc tính và hàm cần thiết cho các lớp đối tượng: hàm quản lý lớp, hàm thực hiện cài đặt lớp, hàm truy nhập vào lớp, hàm xử lý lỗi
- B5: Thiết kế các hàm thành phần của lớp: sử dụng kỹ thuật phân rã chức năng Top – Down, sử dụng kỹ thuật thiết kế có cấu trúc, kết quả của thiết kế có cấu trúc cho một hàm là một cấu trúc có một lối vào và một lối ra được tổ hợp từ ba cấu trúc cơ bản là tuần tự, tuyển chọn và vòng lặp.
- B6: Thiết kế chương trình chính với các nhiệm vụ:
 - Nhập dữ liệu từ người sử dụng;
 - Tạo ra các đối tượng theo định nghĩa các lớp;
 - Tổ chức thực hiện trao đổi thông tin giữa các đối tượng;
 - Lưu trữ kết quả xử lý hoặc hiện lên màn hình, máy in, thiết bị ngoại vi theo yêu cầu của người, sử dụng.



LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

- Tổ chức chương trình thành các Lớp (Lớp bao gồm dữ liệu và các phương thức xử lý dữ liệu).
- Các cấu trúc dữ liệu được thiết kế sao cho đặc tả được các đối tượng.
- Dữ liệu được bao bọc, che giấu và không cho phép các hàm ngoại lai truy nhập tự do.
- Các đối tượng trao đổi với nhau thông qua các hàm.
- Dễ dàng bổ sung dữ liệu và các hàm mới vào đối tượng khi cần thiết.
- Chương trình được thiết kế theo cách tiếp cận Bottom – Up.

CÁC ƯU ĐIỂM CỦA OOP

- Tính kế thừa
⇒ Loại bỏ những đoạn chương trình lặp lại và mở rộng khả năng sử dụng các lớp đã được xây dựng.
- Tính đóng gói, che dấu thông tin
=> Chương trình không bị thay đổi bởi những đoạn chương trình khác.
- Mô phỏng thế giới thực tốt hơn: ánh xạ các đối tượng của bài toán vào đối tượng của chương trình.
- Hệ thống hướng đối tượng dễ mở rộng, nâng cấp thành hệ thống lớn hơn.



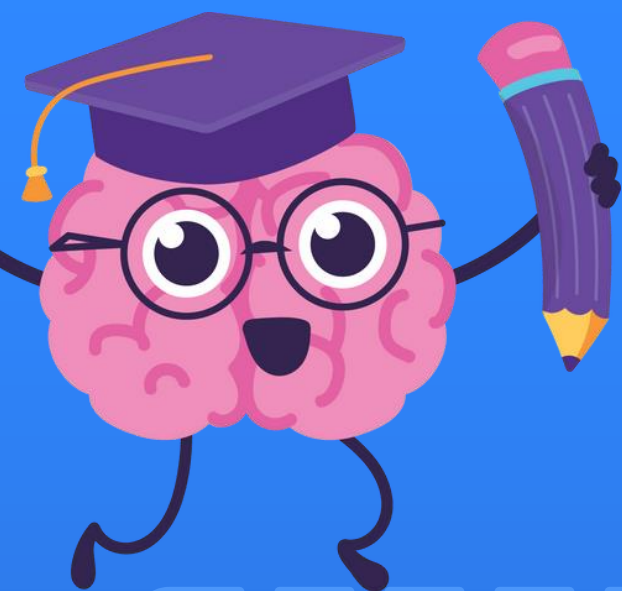
CHƯƠNG III

LỖP

&

ĐỔI TƯỞNG





CHƯƠNG III

LỚP VÀ ĐỐI TƯỢNG

1. Cú pháp khai báo lớp
2. Định nghĩa hàm thành phần của lớp
3. Con trỏ this
4. Khai báo và tạo lập đối tượng
5. Phạm vi truy xuất
6. Phương thức thiết lập – Constructor
7. Phương thức hủy bỏ – Destructor
8. Phương thức Truy vấn, Cập nhật
9. Thành viên tĩnh – static member
10. Khởi tạo một đối tượng, dữ liệu và hàm thành viên tĩnh



CỦ PHÁP KHAI BẢO LỖP



CÚ PHÁP KHAI BÁO LỚP

```
class <Tên_lớp> {  
    <Phạm vi truy cập> <Kiểu dữ liệu> <Tên_thuộc_tính>;  
    <Phạm vi truy cập> <Kiểu dữ liệu> <Tên_phương_thức>(<Danh sách tham số>);  
  
    // ...  
  
public: // (Phần public)  
    // Khai báo các thuộc tính và phương thức truy cập  
    // được từ bên ngoài lớp  
  
protected: // (Phần protected)  
    // Khai báo các thuộc tính và phương thức chỉ truy cập  
    // được từ bên trong lớp và lớp con  
  
private: // (Phần private)  
    // Khai báo các thuộc tính và phương thức chỉ truy cập  
    // được từ bên trong lớp  
};
```

VÍ DỤ

```
class Person {  
public:  
    string name;  
    int age;  
  
private:  
    string address;  
  
public:  
    void introduce() {  
        cout << "Tên: " << name << endl;  
        cout << "Tuổi: " << age << endl;  
    }  
};
```



ĐỊNH NGHĨA HÀM THÀNH PHẦN CỦA LỚP



ĐỊNH NGHĨA HÀM THÀNH PHẦN

- Hàm thành phần (hay còn gọi là phương thức) là một hàm được định nghĩa bên trong một lớp. Nó hoạt động trên dữ liệu của lớp và có thể truy cập trực tiếp các thuộc tính của lớp.

```
<Phạm vi truy cập> <Kiểu trả về> <Tên_hàm>(<Danh sách tham số>) {  
    // Thân hàm (thực thi các lệnh)  
}
```

- <Phạm vi truy cập>: Quy định mức độ truy cập của hàm, tương tự như phạm vi truy cập của thuộc tính.
- public: Truy cập được từ bên ngoài lớp.
- protected: Truy cập được từ bên trong lớp và lớp con.
- private: Truy cập được từ bên trong lớp.

VÍ DỤ

```
class Person {  
public:  
    string name;  
    int age;  
  
public:  
    void introduce() {  
        cout << "Tên: " << name << endl;  
        cout << "Tuổi: " << age << endl;  
    }  
  
private:  
    void changeName(string newName) {  
        name = newName;  
    }  
};
```




GỌI HÀM THÀNH PHẦN CỦA LỚP

- Sử dụng tên hàm thành phần sau tên đối tượng, ngăn cách bởi dấu chấm.

```
class MyClass {
public:
    void myFunction() {
        // ...
    }
};

int main() {
    MyClass obj;

    // Gọi hàm thành phần "myFunction"
    // của đối tượng "obj"
    obj.myFunction();

    return 0;
}
```

GỌI HÀM THÀNH PHẦN CỦA LỚP

- Sử dụng toán tử -> để truy cập thành viên của lớp thông qua con trỏ.

```
class MyClass {
public:
    void myFunction() {
        // ...
    }
};

int main() {
    MyClass obj;
    MyClass* ptr = &obj;

    // Gọi hàm thành phần "myFunction"
    // của đối tượng "obj" thông qua con trỏ "ptr"
    ptr->myFunction();

    return 0;
}
```



CON TRỎ THIS



CON TRỎ THIS

- Từ khóa this trong định nghĩa của các hàm thành phần lớp dùng để xác định địa chỉ của đối tượng dùng làm tham số ngầm định cho hàm thành phần.
- Con trỏ this tham chiếu đến đối tượng đang gọi Hàm thành phần.

CON TRỎ THIS	CON TRỎ THÔNG THƯỜNG
Được tạo tự động bởi trình biên dịch.	Được khai báo và khởi tạo bởi người lập trình
Luôn trỏ đến chính đối tượng đang được thực thi	Có thể trỏ đến bất kỳ đối tượng nào.
Không cần giải phóng bộ nhớ.	Cần giải phóng bộ nhớ khi không sử dụng nữa.

VÍ DỤ

```
class SinhVien {  
public:  
    string hoTen;  
    int tuoi;  
  
    void inThongTin() {  
        cout << "Ho ten: " << this->hoTen << endl;  
        cout << "Tuoi: " << this->tuoi << endl;  
    }  
};
```



KHAI BẢO & TẠO LẬP ĐỐI TƯỢNG



KHAI BÁO ĐỐI TƯỢNG

- Khai báo đối tượng:
<tên lớp> <tên đối tượng>;
Ví dụ: DIEM d1, d2;
- Khai báo mảng đối tượng:
<tên lớp> <tên mảng đối tượng>[số phần tử];
Ví dụ: DIEM d[20];
⇒ d là mảng kiểu DIEM gồm 20 phần tử.
- Mỗi đối tượng sau khi khai báo sẽ được cấp phát một vùng nhớ riêng để chứa các thuộc tính của chúng.
- Không có vùng nhớ riêng để chứa các phương thức cho mỗi đối tượng.

KHAI BÁO ĐỐI TƯỢNG

- Các phương thức sẽ được sử dụng chung cho tất cả đối tượng cùng lớp.
- Truy xuất thuộc tính của đối tượng:

```
Tên_Đối_Tượng.Tên_Thuộc_Tính  
Tên_Mảng_Đối_Tượng[chỉ_số].Tên_Thuộc_Tính
```

- Sử dụng các phương thức của lớp (mà đối tượng thuộc về):

```
Tên_Đối_Tượng.Tên_Phương_Thức(ds_đối);  
Tên_Mảng_Đối_Tượng[chỉ_số].Tên_Phương_Thức(ds_đối);
```

=> Để chỉ rõ phương thức thực hiện trên các thuộc tính của đối tượng nào.



PHẠM VI TRUYỀN XUẤT



PHẠM VI TRUY XUẤT

1. Public

- Thuộc tính và phương thức public có thể được truy cập trực tiếp từ bất kỳ nơi nào trong chương trình, bao gồm:

- Bên trong lớp
- Lớp con
- Các hàm bên ngoài lớp

Ví dụ:

```
class SinhVien {  
public:  
    string hoTen; // Thuộc tính public  
    int tuoi; // Thuộc tính public  
  
    void inThongTin() { // Phương thức public  
        cout << "Ho ten: " << hoTen << endl;  
        cout << "Tuoi: " << tuoi << endl;  
    }  
};  
  
int main() {  
    SinhVien sv;  
    sv.hoTen = "Nguyen Van A"; // Truy cập trực tiếp thuộc tính public  
    sv.tuoi = 20;  
  
    sv.inThongTin(); // Gọi phương thức public  
  
    return 0;  
}
```



PHẠM VI TRUY XUẤT

1. Protected

- Thuộc tính và phương thức protected chỉ truy cập được từ:
 - o Bên trong lớp
 - o Lớp con
- Không thể truy cập trực tiếp từ các hàm bên ngoài lớp.
- Mục đích: che giấu chi tiết triển khai bên trong lớp, chỉ cho phép truy cập và sửa đổi bởi lớp con.

Ví dụ:

```
class SinhVien {
protected:
    string diaChi; // Thuộc tính protected

    void setDiaChi(string diaChi) { // Phương thức protected
        this->diaChi = diaChi;
    }
public:
    void inThongTin() {
        cout << "Dia chi: " << diaChi << endl;
    }
};

class SinhVienDaiHoc : public SinhVien {
public:
    void setThongTin(string hoTen, int tuoi, string diaChi) {
        this->hoTen = hoTen;
        this->tuoi = tuoi;
        setDiaChi(diaChi); // Truy cập và sửa đổi thuộc tính protected từ lớp con
    }
};

int main() {
    SinhVienDaiHoc sv;
    // sv.diaChi = "Ha Noi"; // Lỗi truy cập trực tiếp thuộc tính protected từ bên ngoài lớp
    sv.setThongTin("Nguyen Van B", 21, "Ha Noi");
    sv.inThongTin();
    return 0;
}
```



PHẠM VI TRUY XUẤT

1. Private

- Thuộc tính và phương thức private chỉ truy cập được từ bên trong lớp.
- Không thể truy cập trực tiếp từ lớp con hay các hàm bên ngoài lớp.
- Mục đích: che giấu chi tiết triển khai và bảo vệ dữ liệu nội bộ của lớp

Ví dụ:

```
class SinhVien {  
private:  
    string maSo; // Thuộc tính private  
  
    void setMaSo(string maSo) { // Phương thức private  
        this->maSo = maSo;  
    }  
  
public:  
    void inThongTin() {  
        cout << "Ma so: " << maSo << endl;  
    }  
};  
  
int main() {  
    SinhVien sv;  
    // sv.maSo = "SV1234"; // Lỗi truy cập trực tiếp thuộc tính private từ bên ngoài lớp  
    // sv.setMaSo("SV1234"); // Lỗi truy cập trực tiếp phương thức private từ bên ngoài lớp  
  
    return 0;  
}
```



PHƯƠNG THỨC THIẾT LẬP



PHƯƠNG THỨC THIẾT LẬP

- Khái niệm: Phương thức thiết lập (hay hàm thiết lập) là một phương thức đặc biệt được gọi tự động khi một đối tượng được tạo ra.
- Mục đích: Khởi tạo các giá trị ban đầu cho các thuộc tính của đối tượng.
- Các loại phương thức thiết lập:
 - Hàm thiết lập mặc định: Không có tham số, khởi tạo các giá trị mặc định cho các thuộc tính.
 - Hàm thiết lập có tham số: Khởi tạo các giá trị ban đầu cho các thuộc tính dựa trên các tham số được truyền vào.

Ví dụ:

```
class SinhVien {  
public:  
    string hoTen;  
    int tuoi;  
  
    SinhVien() {} // Hàm khởi tạo mặc định  
  
    SinhVien(string hoTen, int tuoi) {  
        this->hoTen = hoTen;  
        this->tuoi = tuoi;  
    } // Hàm khởi tạo có tham số  
};
```



KHAI BÁO, ĐỊNH NGHĨA

- Khai báo:

```
class <Tên_lớp> {  
public:  
    // Khai báo các phương thức khởi tạo  
};
```

- Định nghĩa

```
class SinhVien {  
public:  
    SinhVien() {}  
  
    SinhVien(string hoTen, int tuoi) {  
        this->hoTen = hoTen;  
        this->tuoi = tuoi;  
    }  
};
```

LƯU Ý

- Mỗi lớp có thể có một hoặc nhiều hàm khởi tạo.
- Tên của hàm khởi tạo phải trùng với tên của lớp.
- Hàm khởi tạo không có kiểu trả về.
- Khi sử dụng toán tử new để tạo đối tượng, hàm khởi tạo sẽ tự động được gọi.
- Trong một lớp có thể có nhiều hàm tạo (các hàm này cùng tên nhưng khác danh sách đối).



PHƯƠNG THỨC HỦY BỎ



PHƯƠNG THỨC HỦY BỎ

- Khái niệm: Phương thức hủy bỏ (hay hàm hủy bỏ) là một phương thức đặc biệt được gọi tự động khi một đối tượng bị hủy. (dọn dẹp 1 đối tượng trước khi nó được thu hồi)
- Mục đích: Giải phóng bộ nhớ được cấp phát cho đối tượng.
- Cú pháp: Phương thức Destructor có tên trùng tên với tên lớp và có dấu ~ đặt trước
- Lưu ý:
 - Destructor phải có thuộc tính public
 - Không có giá trị trả về và không thể định nghĩa lại (nó không bao giờ có tham số). Mỗi lớp chỉ có 1 destructor.
 - Không gọi trực tiếp, sẽ được tự động gọi khi hủy bỏ đối tượng.
 - Nếu ta không cung cấp destructor, C++ sẽ tự sinh một destructor rỗng (không làm gì cả).

```
class SinhVien {  
public:  
    string hoTen;  
    int tuoi;  
  
    SinhVien() {}  
  
    ~SinhVien() {  
        // Giải phóng bộ nhớ được cấp phát cho `hoTen`  
        delete[] hoTen;  
    }  
  
    // ...  
};  
  
int main() {  
    SinhVien* sv = new SinhVien();  
    sv->hoTen = "Nguyen Van A";  
    sv->tuoi = 20;  
  
    // ...  
  
    // Xóa đối tượng và giải phóng bộ nhớ  
    delete sv;  
  
    return 0;  
}
```



PHƯƠNG THỨC TRUY VẤN CẬP NHẬT



PHƯƠNG THỨC TRUY VẤN, CẬP NHẬT

- Phương thức truy vấn: còn gọi là getter, dùng để truy cập giá trị của thuộc tính (biến thành viên) private trong class.
- Phương thức cập nhật: còn gọi là setter, dùng để thay đổi giá trị của thuộc tính private trong class.
- Lưu ý:
 - Tên phương thức: Getter thường bắt đầu bằng "get", setter thường bắt đầu bằng "set".
 - Kiểu trả về: Getter thường trả về kiểu dữ liệu của thuộc tính. Setter thường không có kiểu trả về.
 - Từ khóa const: Getter thường được khai báo với từ khóa const để đảm bảo không thay đổi giá trị của thuộc tính.

```
class SinhVien {  
private:  
    string hoTen;  
    int tuoi;  
  
public:  
    // Phương thức truy vấn  
    string getHoTen() const { return hoTen; }  
    int getTuoi() const { return tuoi; }  
  
    // Phương thức cập nhật  
    void setHoTen(string hoTen) { this->hoTen = hoTen; }  
    void setTuoi(int tuoi) { this->tuoi = tuoi; }  
};
```



THÀNH VIÊN TĨNH



THÀNH VIÊN TĨNH

- Thành viên tĩnh: là thuộc tính hoặc phương thức thuộc về class chứ không thuộc về các đối tượng của class.
- Đặc điểm:
 - Chỉ có một bản sao duy nhất của thành viên tĩnh cho tất cả các đối tượng của class.
 - Không thể truy cập trực tiếp thành viên tĩnh thông qua đối tượng.
 - Có thể truy cập thành viên tĩnh thông qua tên class và toán tử phân giải phạm vi ::
- Ứng dụng:
Đếm số lượng đối tượng của class đã được tạo.

THÀNH VIÊN TĨNH

- Lưu trữ dữ liệu chung cho tất cả các đối tượng của class.
Cung cấp các hàm tiện ích không liên quan đến trạng thái của một đối tượng cụ thể.
- Lưu ý:
 - Khởi tạo: Biến tĩnh cần được khởi tạo trước khi sử dụng.
 - Truy cập:
 - Có thể truy cập trực tiếp thành viên tĩnh trong class.
 - Có thể truy cập thành viên tĩnh thông qua đối tượng nhưng không khuyến khích.



THÀNH VIÊN TĨNH

Ví dụ:

```
class SinhVien {
public:
    static int soLuongSinhVien; // Biến tĩnh

    SinhVien() {
        soLuongSinhVien++;
    }

    ~SinhVien() {
        soLuongSinhVien--;
    }

    // ...
};

int SinhVien::soLuongSinhVien = 0; // Khởi tạo biến tĩnh

int main() {
    SinhVien sv1;
    SinhVien sv2;

    // Truy cập biến tĩnh
    cout << "Số lượng sinh viên: " << SinhVien::soLuongSinhVien << endl;

    return 0;
}
```



KHỞI TẠO ĐỐI TƯỢNG, DỮ LIỆU VÀ HÀM THÀNH VIÊN TĨNH



KHỞI TẠO ĐỐI TƯỢNG, DỮ LIỆU VÀ HÀM THÀNH VIÊN TĨNH

```
class SinhVien {
public:
    static int soLuongSinhVien; // Biến tĩnh

    SinhVien() {
        soLuongSinhVien++;
    }

    ~SinhVien() {
        soLuongSinhVien--;
    }

    // ...
};

int SinhVien::soLuongSinhVien = 0; // Khởi tạo biến tĩnh

int main() {
    SinhVien sv1;
    SinhVien sv2;

    // Truy cập và cập nhật biến tĩnh
    cout << "Số lượng sinh viên: " << SinhVien::soLuongSinhVien << endl;

    return 0;
}
```

```
class SinhVien {
public:
    static void inThongTin() {
        cout << "Đây là class SinhVien" << endl;
    }

    // ...
};

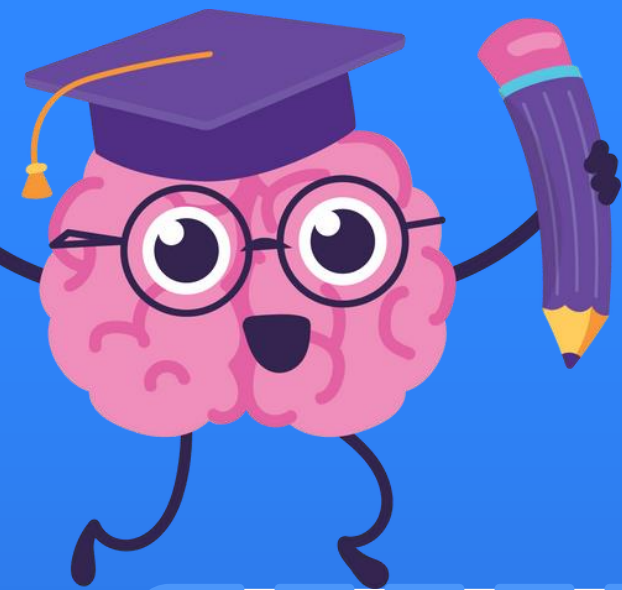
int main() {
    // Gọi hàm thành viên tĩnh
    SinhVien::inThongTin();

    return 0;
}
```



CHƯƠNG IV VIỆC KHỞI TẠO ĐỐI TƯỢNG, HÀM BẠN, LỚP BẠN





CHƯƠNG IV

VIỆC KHỞI TẠO ĐỐI TƯỢNG, HÀM BẠN, LỚP BẠN

1. Đối tượng là thành phần của lớp
2. Đối tượng là thành phần của mảng
3. Đối tượng được cấp phát động
4. Hàm bạn
5. Lớp bạn
6. Một số nguyên tắc xây dựng lớp



**Đối tượng là
thành phần
của lớp**



Đối tượng là thành phần của lớp

1. Đối tượng có thể là thành phần của đối tượng khác:

- Giống như các kiểu dữ liệu cơ bản, đối tượng cũng có thể được sử dụng để tạo ra các đối tượng khác.
- Điều này giúp tạo ra các cấu trúc dữ liệu phức tạp hơn từ các đơn giản hơn.

2. Vòng đời của các đối tượng thành phần:

- Khi một đối tượng "lớn" (đối tượng kết hợp) được tạo ra, các đối tượng thành phần của nó cũng được tạo ra.
- Khi đối tượng kết hợp bị hủy, các đối tượng thành phần của nó cũng bị hủy.
- Phương thức hủy bỏ sẽ được gọi cho các đối tượng thành phần sau khi phương thức hủy bỏ của đối tượng kết hợp được gọi.

3. Cung cấp tham số cho các phương thức thiết lập của đối tượng thành phần:

- Nếu phương thức thiết lập của đối tượng thành phần cần cung cấp tham số, thì đối tượng kết hợp phải có phương thức thiết lập để cung cấp tham số cho các phương thức thiết lập này.
- Dấu hai chấm (:) được sử dụng để cung cấp tham số cho các phương thức thiết lập của đối tượng thành phần, tiếp theo là tên đối tượng thành phần và các tham số khởi tạo được truyền vào.



Đối tượng là thành phần của lớp

Ví dụ:

Lớp Car có thể có các thành phần Engine và Tire. Khi một đối tượng Car được tạo ra, các đối tượng Engine và Tire cũng được tạo ra. Khi đối tượng Car bị hủy, các đối tượng Engine và Tire cũng bị hủy.

Đối tượng là thành phần của lớp

```
class Engine {
public:
    Engine() {}
    ~Engine() {}
};

class Tire {
public:
    Tire() {}
    ~Tire() {}
};

class Car {
public:
    Car() {
        engine = new Engine();
        tire = new Tire();
    }

    ~Car() {
        delete engine;
        delete tire;
    }

private:
    Engine* engine;
    Tire* tire;
};
```



ĐỐI TƯỢNG LÀ THÀNH PHẦN CỦA MẢNG



ĐỐI TƯỢNG LÀ THÀNH PHẦN CỦA MẢNG

1. Mảng và khởi tạo:

- Khi một mảng được tạo ra, các phần tử của nó cũng được tạo ra.
- Phương thức thiết lập sẽ được gọi cho từng phần tử.

2. Vấn đề khởi tạo:

- Không thể cung cấp tham số khởi tạo cho tất cả các phần tử của mảng.
- Do đó, mỗi đối tượng trong mảng phải có khả năng tự khởi tạo, nghĩa là có thể thiết lập không cần tham số.

3. Các trường hợp tự khởi tạo:

- Lớp không có phương thức thiết lập
- Lớp có phương thức thiết lập không tham số
- Lớp có phương thức thiết lập mà mọi tham số đều có giá trị mặc nhiên



ĐỔI TƯỢNG ĐƯỢC CẬP PHÁT ĐỘNG



ĐỐI TƯỢNG ĐƯỢC CẤP PHÁT ĐỘNG

- Sử dụng toán tử new để cấp phát đối tượng trong class.

```
class MyClass {  
public:  
    int x;  
    MyClass() {  
        cout << "Constructor được gọi" << endl;  
    }  
    ~MyClass() {  
        cout << "Destructor được gọi" << endl;  
    }  
};  
  
int main() {  
    MyClass* obj = new MyClass(); // Cấp phát đối tượng  
    obj->x = 10; // Truy cập và sửa đổi thuộc tính  
    delete obj; // Hủy đối tượng  
    return 0;  
}
```

- Khi sử dụng new để cấp phát đối tượng trong class, trình biên dịch sẽ thực hiện:

- Cấp phát vùng nhớ trong heap cho đối tượng.
- Gọi constructor của class để khởi tạo đối tượng.

- Khi sử dụng delete để hủy đối tượng, trình biên dịch sẽ thực hiện:

- Gọi destructor của class để giải phóng tài nguyên của đối tượng.
- Giải phóng vùng nhớ được cấp phát cho đối tượng.



ĐỐI TƯỢNG ĐƯỢC CẬP PHÁT ĐỘNG

Cấp phát:

- Sử dụng toán tử `new []` với số lượng phần tử mong muốn trong ngoặc vuông.

Ví dụ:

```
MyClass* arr = new MyClass[10];  
// Cấp phát mảng 10 phần tử kiểu MyClass
```

Xóa cấp phát:

- Sử dụng toán tử `delete []` với dấu ngoặc vuông trước tên biến.

Ví dụ:

```
delete[] arr;  
// Hủy mảng arr được cấp phát ở trên
```

```
class MyClass {  
public:  
    int x;  
    MyClass() {  
        cout << "Constructor được gọi" << endl;  
    }  
    ~MyClass() {  
        cout << "Destructor được gọi" << endl;  
    }  
};  
  
int main() {  
    MyClass* arr = new MyClass[10];  
    // Cấp phát mảng 10 phần tử  
    for (int i = 0; i < 10; i++) {  
        arr[i].x = i;  
        // Truy cập và sửa đổi thuộc tính  
    }  
    delete[] arr; // Hủy mảng  
    return 0;  
}
```



HÀM BẠN



HÀM BẠN

1. Khái niệm

- Hàm bạn (friend function) là một hàm có thể truy cập trực tiếp vào các thành viên private và protected của một lớp mà không cần là thành viên của lớp đó.
- Cú pháp: khai báo prototype của hàm trong lớp muốn cho phép truy cập đó với từ khóa friend.

```
class MyClass {  
private:  
    int x;  
  
public:  
    friend void printX(MyClass& obj);  
};  
  
void printX(MyClass& obj) {  
    // Truy cập trực tiếp vào thành viên private `x`  
    cout << obj.x << endl;  
}
```

1. Đặc điểm

- Được khai báo bên ngoài lớp: Hàm bạn được khai báo bên ngoài lớp mà nó muốn truy cập.
- Từ khóa friend: Sử dụng từ khóa friend để khai báo hàm bạn.
- Truy cập trực tiếp: Hàm bạn có thể truy cập trực tiếp vào các thành viên private và protected của lớp.
- Không phải thành viên: Hàm bạn không phải là thành viên của lớp và không có quyền truy cập vào các thành viên private và protected của các lớp khác.



HÀM ĐỘC LẬP LÀ HÀM BẠN

- Hàm độc lập (không thuộc lớp nào) có thể được khai báo là hàm bạn của một lớp.
- Chỉ cần khai báo tên hàm độc lập bên trong lớp.
- Trong nội dung hàm độc lập, có thể truy cập bất kỳ thành phần nào thuộc lớp đó.

```
class PhanSo {  
    int tu, mau;  
public:  
    ...  
    friend int SoSanhBang(PhanSo, PhanSo);  
};  
int SoSanhBang(PhanSo a, PhanSo b) {  
    if (a.tu * b.mau == b.tu * a.mau)  
        return 1;  
    else  
        return 0;  
}
```



HÀM THÀNH VIÊN LÀ HÀM BẠN

- Hàm thành viên của một lớp có thể được khai báo là bạn của một lớp khác.
- Chỉ cần khai báo:
<Tên lớp>::<Tên hàm thành viên> bên trong lớp cần truy cập.
- Trong nội dung hàm thành viên, có thể truy cập bất kỳ thành phần nào thuộc lớp đã khai báo.
- Chú ý:
 - + Một lớp có thể có nhiều hàm bạn (độc lập hay hàm thành viên của lớp khác).
 - + Một hàm (độc lập hay hàm thành viên) có thể là bạn của nhiều lớp.
 - + Hàm bạn đã phá vỡ tính đóng gói của OOP => không lạm dụng.

```
class DoanThang {  
    Diem d1, d2;  
public:  
    ...  
    float ChieuDai() {  
        sqrt(  
            pow((d1.x - d2.x), 2)  
            +  
            pow((d1.y - d2.y), 2));  
    }  
};  
class Diem {  
    int x, y;  
public:  
    ...  
    friend float  
        DoanThang::ChieuDai();  
};
```




BAN HỌC TẬP
HỆ THỐNG THÔNG TIN



LỚP BẠN



MỘT SỐ LƯU Ý

- Nếu lớp A được khai báo là bạn của lớp B (trong định nghĩa của lớp B chứa câu lệnh “friend class A;”) thì tất cả hàm thành phần của lớp A đều có thể truy nhập đến các thành phần private của lớp B.
- Một lớp có thể là bạn của nhiều lớp khác.
- Trong lớp B có thể khai báo lớp A là bạn và trong lớp A có thể khai báo lớp B là bạn.
- Các tính chất của quan hệ friend:
 - Không đối xứng
 - Không bắc cầu

VÍ DỤ

```
#include <bits/stdc++.h>
using namespace std;

class B {
private:
    int b;
public:
    B()
    {
        b = 10;
    }
    friend class A;
};

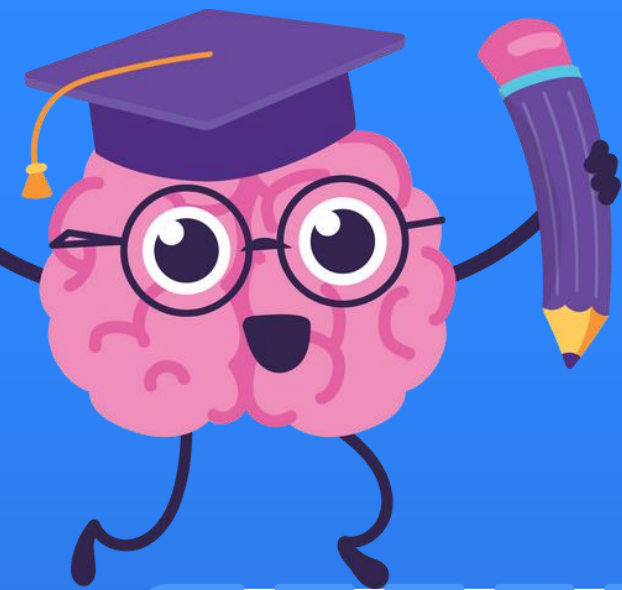
class A {
public:
    void print(B arg)
    {
        cout << arg.b;
    }
};

int main()
{
    B b;
    A a;
    a.print(b);
    return 0;
}
```



CHƯƠNG V OVERLOAD TOÁN TỬ





CHƯƠNG V

OVERLOAD TOÁN TỬ

1. Các toán tử của C++
2. Overload toán tử
3. Chuyển kiểu
4. Sự nhập nhằng
5. Overload một số toán tử thông dụng
6. Phép toán tăng và giảm: ++ và --



Các toán tử của C++



CÁC LOẠI TOÁN TỬ

- Operator: toán tử
- Operation: phép toán
- Operand: toán hạng
- Expression: biểu thức
- Toán tử là các ký hiệu được dùng để thực hiện một phép toán trong ngôn ngữ lập trình.
- Toán tử thao tác trên hằng hoặc biến, hằng hoặc biến này được gọi là toán hạng.
- Một biểu thức là tổ hợp các toán tử và toán hạng, một biểu thức sẽ được tính toán để cho ra một giá trị.
- Phân loại toán tử:
 - Toán tử một ngôi: Chỉ có một toán hạng. Ví dụ: $++x$, $--y$.
 - Toán tử hai ngôi: Có hai toán hạng. Ví dụ: $x + y$, $a * b$.
 - Toán tử ba ngôi: Có ba toán hạng. Ví dụ: $x ? y : z$.



PHÂN LOẠI TOÁN TỬ

- 1) Toán tử số học: $+$ $-$ $*$ $/$ $\%$ (chỉ dùng cho số nguyên) $++$ $--$
- 2) Toán tử quan hệ: $<$ $<=$ $>$ $>=$ $==$ $!=$
- 3) Toán tử logic: $\&\&$ $\|\$ $!(\text{not})$
- 4) Toán tử gán: $=$ $+=$ $-=$ $*=$ $/=$ $\%=$
- 5) Toán tử thao tác trên bit: $\&$ $|$ $^$ (XOR) \sim (đảo bit) $>>$ $<<$ và $\&=$ $|=$ $\wedge=$ $>>=$ $<<=$
(ví dụ $A \&= 2 \Rightarrow A = A \& 2$)
- 6) Toán tử làm việc với con trỏ: $\&$ $*$
- 7) Toán tử khác: $()$ // gọi hàm
 $[]$ // truy xuất phần tử mảng



OVERLOAD TOÁN TỬ



CÁC LOẠI TOÁN TỬ

Các phép toán có thể tái định nghĩa:

Đơn hạng	+	-	*	!	~	&	++	--	()	->	->*
	new	delete									
Nhị hạng	+	-	*	/	%	&		^	<<	>>	
	=	+=	-=	/=	%=	&=	=	^=	<<=	>>=	
	==	!=	<	>	<=	>=	&&		[]	()	,

- Các phép toán không thể tái định nghĩa:

. * :: ?: sizeof



CÚ PHÁP

- Cú pháp:

Kiểu_trả_về operator Tên_toán_tử
(danh sách đối số)

- Số lượng đối số của hàm toán tử phụ thuộc vào:

- Toán tử đơn hay toán tử đôi;
 - Hàm toán tử là hàm thành phần của lớp hay hàm
- toàn cục.

VÍ DỤ

```
class Complex {  
public:  
    Complex(double real, double imag) : real(real), imag(imag) {}  
  
    Complex operator + (const Complex& other) {  
        return Complex(real + other.real, imag + other.imag);  
    }  
  
private:  
    double real;  
    double imag;  
};
```




HẠN CHẾ CỦA OVERLOAD TOÁN TỬ

- Không thể overload một toán tử mà chưa từng được định nghĩa trước đó, hay nói cách khác là không thể tạo toán tử mới.
- Không thể tạo cú pháp mới cho toán tử.
- Không thể thay đổi định nghĩa có sẵn của một toán tử.
- Không thể thay đổi thứ tự ưu tiên của các toán tử.

MỘT SỐ LƯU Ý

- Các hàm toán tử sau phải là hàm thành phần của lớp, khi đó đối tượng gọi hàm sẽ là toán hạng thứ nhất của toán tử.
 - operator =
 - operator []
 - operator ()
 - operator ->
- Nếu có sử dụng các toán tử gán ($+=$, $-=$, $*=$, ...) thì chúng phải được định nghĩa (là các hàm toán tử), cho dù đã định nghĩa toán tử gán ($=$) và các toán tử số học ($+$, $-$, $*$, ...) cho lớp.
- Cài đặt hàm toán tử đúng ý nghĩa của toán tử được overload.



OVERLOAD BẰNG HÀM THÀNH VIÊN

- Khi đa năng hóa (), [], -> hoặc =, hàm đa năng hóa toán tử phải được khai báo như một thành viên lớp;
- Toán tử một ngôi hàm không có tham số, toán tử 2 ngôi hàm sẽ có 1 tham số.

VÍ DỤ

```
#include <iostream>
using namespace std;

class Point {
public:
    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
    Point operator+(Point& other) {
        return Point(x + other.x, y + other.y);
    }
    Point operator-(Point& other) {
        return Point(x - other.x, y - other.y);
    }
private:
    int x;
    int y;
};

int main() {
    Point p1(10, 20), p2(20, 10);
    Point p3 = p1 + p2;
    Point p4 = p1 - p2;
    Point p5 = p3.operator+(p4);
    Point p6 = p3.operator-(p4);
    return 0;
}
```



OVERLOAD BẰNG HÀM TOÀN CỤC

- Đa năng hóa toán tử bằng hàm toàn cục: nếu toán hạng cực trái của toán tử là đối tượng thuộc lớp khác hoặc thuộc kiểu dữ liệu có sẵn.

=> Thường khai báo friend

VÍ DỤ

```
#include <iostream>
#include <vector>
using namespace std;

class Point {
public:
    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
    friend Point operator+(Point& p1, Point& p2);
    friend Point operator-(Point& p1, Point& p2);
private:
    int x;
    int y;
};

Point operator+(Point& p1, Point& p2) {
    return Point(p1.x + p2.x, p1.y + p2.y);
}

Point operator-(Point& p1, Point& p2) {
    return Point(p1.x - p2.x, p1.y - p2.y);
}

int main() {
    Point p1(10, 20), p2(20, 10);

    Point p3 = p1 + p2;
    Point p4 = p1 - p2;
    Point p5 = operator+(p1, p2);
    Point p6 = operator-(p1, p2);

    return 0;
}
```



CHUYỂN KIỂU



CHUYỂN KIỂU BẰNG CONSTRUCTOR

- Ta dùng cách chuyển kiểu bằng phương thức thiết lập khi thỏa hai điều kiện sau:
- Chuyển từ kiểu đã có sang kiểu đang định nghĩa.
- Có quan hệ “là một” giữa kiểu đã có và kiểu đang định nghĩa (ví dụ một số nguyên là một phân số).
- Chuyển kiểu bằng Constructor có nhược điểm:
- Không thể chuyển từ kiểu dữ liệu tự định nghĩa sang kiểu dữ liệu chuẩn, vì không thể sửa đổi kiểu dữ liệu chuẩn.
- Phương thức thiết lập với một tham số sẽ dẫn đến cơ chế chuyển kiểu tự động có thể không mong muốn.

VÍ DỤ

```
class PhanSo {  
public:  
    PhanSo(int tu, int mau) {  
        this->tu = tu;  
        this->mau = mau;  
    }  
private:  
    int tu;  
    int mau;  
};  
  
int main() {  
    PhanSo ps(2, 3); // Chuyển kiểu từ int sang PhanSo  
    return 0;  
}
```




CHUYỂN KIỂU BẰNG TỰ ĐỊNH NGHĨA

- Phép toán chuyển kiểu là hàm thành phần có dạng:

Tên_lớp::operator Kiểu_dữ_liệu_chuẩn()

=> Sẽ có cơ chế chuyển kiểu tự động từ kiểu dữ liệu tự định nghĩa (Tên_lớp) sang kiểu dữ liệu chuẩn.

- Ta dùng phép toán chuyển kiểu khi định nghĩa kiểu dữ liệu mới và muốn sử dụng các phép toán đã có của kiểu dữ liệu chuẩn.
- Phép toán chuyển kiểu cũng được dùng để biểu diễn quan hệ “là một” giữa kiểu đang định nghĩa và kiểu đã có (ví dụ một phân số là một số thực).

VÍ DỤ

```
class PhanSo {
public:
    operator double() const {
        return (double)tu / mau;
    }
private:
    int tu;
    int mau;
};

int main() {
    PhanSo ps(2, 3);
    double d = ps; // Chuyển kiểu từ PhanSo sang double
    return 0;
}
```



SỰ NHẬP NHẢNG



SỰ NHẬP NHẲNG

- Sự nhập nhằng xảy ra khi trình biên dịch tìm được ít nhất hai cách chuyển kiểu để thực hiện một tính toán nào đó.
- Hiện tượng này thường xảy ra khi người sử dụng định nghĩa lớp và qui định cơ chế chuyển kiểu bằng phương thức thiết lập và/hay phép toán chuyển kiểu.

VÍ DỤ

```
void main() {  
    PhanSo a(2, 3), b(3, 4), c;  
  
    c = a + 2; c = 2 + a;  
    /*Nhập nhằng: chuyển a sang số thực hay là chuyển 2 sang phân  
    số??? -> báo lỗi: "more than one operator "+" matches these  
    operands" => Bỏ phép toán chuyển kiểu, khi đó trình biên dịch sẽ  
    chuyển 2 sang phân số */  
  
    double r = 2.5 + a; r = a + 2.5;  
    /*Báo lỗi tương tự vì có sự nhập nhằng: chuyển 2.5 sang phân số hay  
    là chuyển a sang số thực??? => Bỏ hàm toán tử +, khi đó trình biên  
    dịch sẽ chuyển a sang số thực */  
}
```



SỰ NHẬP NHẢNG

- Để tránh sự nhập nhằng ta phải chuyển kiểu một cách tường minh, không áp dụng cơ chế chuyển kiểu tự động, mặc dù điều này làm mất đi sự tiện lợi của cơ chế chuyển kiểu tự động.

```
struct PhanSo {  
  
}  
  
void main() {  
    PhanSo a(2, 3), b(3, 4), c;  
    c = a + b;  
    c = a + PhanSo(2);  
    c = PhanSo(2) + a;  
    double r = 2.5 + double(a);  
    r = double(a) + 2.5;  
    cout << r << "\n";  
}
```



Overload một số toán tử thông dụng



TOÁN TỬ GÁN

- Khi lớp có các thuộc tính kiểu con trỏ hay tham chiếu thì không sử dụng hàm tạo sao chép mặc định và toán tử gán mặc định được bởi vì khi đó 2 con trỏ (thuộc tính của 2 đối tượng) cùng trỏ đến một vùng nhớ nên sự thay đổi của đối tượng này sẽ ảnh hưởng đến đối tượng kia.

=> Xây dựng Hàm tạo sao chép và Hàm toán tử gán cho lớp.





HÀM TẠO SAO CHÉP

- Hàm tạo sao chép có một đối kiểu tham chiếu để khởi gán cho đối tượng mới.

```
Tên_lớp(const Tên_lớp& u) {  
    this->Tên_thuộc_tính = u.Tên_thuộc_tính;  
    this->Tên_con_trỏ = new Kiểu_dữ_liệu[n];  
    memcpy(this->Tên_con_trỏ, u.Tên_con_trỏ, n);  
}  
Tên_lớp u; //Gọi tới hàm tạo mặc định  
Tên_lớp v(u); //Gọi tới hàm tạo sao chép
```

- Mục đích của hàm tạo sao chép là tạo ra đối tượng v có nội dung ban đầu giống như đối tượng u nhưng độc lập với u.

HÀM TOÁN TỬ GÁN

- Trong hàm toán tử gán, đối ẩn (con trỏ this) biểu thị đối tượng đích và 1 đối tượng minh biểu thị đối tượng nguồn.

```
Kiểu_trả_về operator=(const Tên_lớp& u) {  
    this->Tên_thuộc_tính = u.Tên_thuộc_tính;  
    this->Tên_con_trỏ = new Kiểu_dữ_liệu[n];  
    memcpy(this->Tên_con_trỏ, u.Tên_con_trỏ, n);  
}
```

- Ví dụ:
 - HT v; //gọi tới hàm tạo mặc định của lớp HT
 - v = u; //gọi tới hàm toán tử gán để gán u cho v



HÀM TẠO SAO CHÉP

- Hàm tạo sao chép được dùng để tạo một đối tượng mới và gán nội dung của một đối tượng đã tồn tại cho đối tượng mới vừa tạo.
- Câu lệnh new và câu lệnh khai báo -> gọi hàm tạo:
 - `HT *h = new HT(50,6);` //gọi hàm tạo có đối
 - `HT k = *h;` //kg gọi hàm toán tử gán mà gọi hàm tạo sao chép

HÀM TOÁN TỬ GÁN

- Hàm toán tử gán không tạo ra đối tượng mới, chỉ thực hiện phép gán giữa hai đối tượng đã tồn tại.
- Câu lệnh gán -> gọi hàm toán tử gán:
 - `HT u; u = *h;` //gọi hàm toán tử gán



VÍ DỤ

```
class String {
    char* p;
public:
    String(char* s = "") { p = _strdup(s); }
    String(const String& s) { p = _strdup(s.p); } //Hàm tạo sao chép
    ~String() { cout << "delete " << (void*)p << "\n"; delete[] p; }
    String& operator = (const String& s); //Thêm hàm toán tử gán
    void Output() const { cout << p; }
};

String& String::operator = (const String& s) {
    if (this != &s) {
        delete[] p; //Dọn dẹp tài nguyên cũ
        p = _strdup(s.p);
    } //Sao chép mới
    return *this;
}
```



OVERLOAD TOÁN TỬ « VÀ »



OVERLOAD TOÁN TỬ « VÀ »

- C++ định nghĩa chồng hai toán tử thao tác trên bit là dịch trái << và dịch phải >> trong lớp ostream và istream.
 - Lớp ostream định nghĩa toán tử xuất << (áp dụng cho các kiểu dữ liệu chuẩn) và một số phương thức xuất khác (flush, put, write, ...).
 - Lớp istream định nghĩa toán tử nhập >> (áp dụng cho các kiểu dữ liệu chuẩn) và một số phương thức nhập khác (get, getline, ignore, ...).
- | | |
|---|---|
| <ul style="list-style-type: none">- Dòng cin:• Là một đối tượng kiểu istream• Là dòng nhập chuẩn gắn với bàn phím.• Các thao tác nhập trên dòng cin đồng nghĩa với nhập dữ liệu từ bàn phím. | <ul style="list-style-type: none">- Dòng cout:• Là một đối tượng kiểu ostream• Là dòng xuất chuẩn gắn với màn hình.• Các thao tác xuất trên dòng cout đồng nghĩa với xuất dữ liệu ra màn hình. |
|---|---|



OVERLOAD TOÁN TỬ »

- Định nghĩa chồng toán tử << để nhận dữ liệu từ dòng nhập chuẩn cin và gán cho biến có kiểu tự định nghĩa.

```
friend istream& operator>>(istream& is, Kiểu_tự_đn& a) {  
    is >> a.Thuộc_tính;  
    return is;  
}
```

⇒ Sử dụng: cin >> Đối_tượng_có_kiểu_tự_định_nghĩa;

Nếu không phải là hàm bạn mà là hàm thành phần của lớp thì hàm toán tử trên chỉ có một đối số như sau:

```
istream& operator>>(istream& is) {  
    is >> this.Thuộc_tính; hoặc is >> Dữ_liệu_của_lớp;  
    return is;  
}
```

=> Sử dụng: Đối_tượng_có_kiểu_tự_định_nghĩa.operator>>(cin);



OVERLOAD TOÁN TỬ «

- Định nghĩa chồng toán tử << để gửi ra dòng xuất chuẩn cout dữ liệu có kiểu tự định nghĩa.

```
friend ostream& operator<<(ostream& os, const Kiểu_tự_đn& a)
{
    os << a.Thuộc_tính; //Hàm bạn nên có thể truy xuất dl của lớp
    return os;
}
```

⇒ Sử dụng: cout << Đối_tượng_có_kiểu_tự_định_nghĩa;

Nếu không phải là hàm bạn mà là hàm thành phần của lớp thì hàm toán tử trên chỉ có một đối số như sau:

```
ostream& operator<<(ostream& os) {
    os << this.Thuộc_tính; hoặc os << Dữ_liệu_của_lớp;
    return os;
}
```

=> Sử dụng:

Đối_tượng_có_kiểu_tự_định_nghĩa.operator<<(cout);



VÍ DỤ

```
class PhanSo {
    long tu, mau;
    void UocLuoc();
public:
    void Set(long t, long m);
    PhanSo(long t = 0, long m = 1) { Set(t, m); }
    long LayTu() const { return tu; }
    long LayMau() const { return mau; }
    friend PhanSo operator + (PhanSo a, PhanSo b);
    friend PhanSo operator - (PhanSo a, PhanSo b);
    friend PhanSo operator * (PhanSo a, PhanSo b);
    friend PhanSo operator / (PhanSo a, PhanSo b);
    PhanSo operator -() const { return PhanSo(-tu, mau); }
    friend istream& operator >> (istream& is, PhanSo& p); //nhập phân số
    friend ostream& operator << (ostream& os, PhanSo p); //xuất phân số
};
```

```
istream& operator >> (istream& is, PhanSo& p) {
    is >> p.tu >> p.mau;
    while (!p.mau) {
        cout << "Nhap lai mau so : "; //nhập lại mẫu số nếu mẫu số = 0
        is >> p.mau;
    }
    p.UocLuoc();
    return is;
}

ostream& operator << (ostream& os, PhanSo p) {
    os << p.tu;
    if (p.tu != 0 && p.mau != 1)
        os << "/" << p.mau;
    return os;
}

void main() {
    PhanSo a, b;
    cout << "Nhap phan so a : "; cin >> a;
    cout << "Nhap phan so b : "; cin >> b;
    cout << a << " + " << b << " = " << a + b << "\n";
    cout << a << " - " << b << " = " << a - b << "\n";
    cout << a << " * " << b << " = " << a * b << "\n";
    cout << a << " / " << b << " = " << a / b << "\n";
    system("pause");
}
```



OVERLOAD TOÁN TỬ []



OVERLOAD TOÁN TỬ []

- Thông thường để xuất ra giá trị của 1 phần tử tại vị trí cho trước trong đối tượng;
- Định nghĩa là hàm thành viên;
- Để hàm toán tử [] có thể đặt ở bên trái của phép gán thì hàm phải trả về là một tham chiếu.

VÍ DỤ

```
class Vector
{
private:
    int Size; int* Data;
public:
    Vector(int S = 2, int V = 0);
    ~Vector();
    void Print() const;
    int& operator [] (int i);
};

int& Vector::operator [] (int i)
{
    static int tam = 0;
    return (i > 0) && (i < Size) ? Data[i] : tam;
}

int main()
{
    Vector V(5, 1);
    V.Print();
    for (int i = 0; i < 5; i++)
    {
        V[i] *= (i + 1);
        V.Print();
        V[0] = 10;
        V.Print();
    }
    return 0;
}
```



OVERLOAD TOÁN TỬ GỌI HÀM ()



OVERLOAD TOÁN TỬ []

- Toán tử [] chỉ có một tham số
 - Chỉ được dùng để truy xuất phần tử của đối tượng thuộc loại mảng một chiều.
- Toán tử gọi hàm () có số tham số bất kỳ
 - Được dùng để truy xuất phần tử của các đối tượng thuộc loại mảng hai hay nhiều chiều.

VÍ DỤ

```
class MATRIX {
    float** M;
    int row, col;
public:
    MATRIX(int, int);
    ~MATRIX();
    float& operator() (int, int); //Hàm toán tử ()
};
float MATRIX::operator() (int i, int j) {
    return M[i][j];
}
MATRIX::MATRIX(int r, int c) {
    M = new float* [r];
    for (int i = 0; i < r; i++)
        M[i] = new float[c];
    row = r;
    col = c;
}
~MATRIX::~MATRIX() {
    for (int i = 0; i < col; i++)
        delete[] M[i]; //M[i] = new float[c]
    delete[] M; //M = new float* [r]
}
void main() {
    cout << "Cho ma tran 2x3\n";
    MATRIX a(2, 3);
    int i, j;
    for (i = 0; i < 2; i++)
        for (j = 0; j < 3; j++)
            cin >> a(i, j);
    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 3; j++)
            cout << a(i, j) << " ";
        cout << endl;
    }
}
```



Phép toán tăng và giảm: ++ và --



Phép toán tăng và giảm: ++ và --

- Toán tử ++ (hoặc toán tử --) có 2 loại:
 - Tiền tố: ++n;
 - Hậu tố: n++ (Hàm toán tử ở dạng hậu tố có thêm đối số giả kiểu int).

Tăng trước ++num:

- Giá trị trả về:
 - Trả về tham chiếu (MyNumber &);
 - Giá trị trái - lvalue (có thể được gán trị).

- Prototype

MyNumber& MyNumber::operator++();

Tăng sau num++:

- Giá trị trả về:
 - Trả về giá trị (giá trị cũ trước khi tăng);
 - Trả về đối tượng tạm thời chứa giá trị cũ;
 - Giá trị phải - rvalue (không thể làm đích của phép gán).

- Prototype

const MyNumber MyNumber::operator++(int)



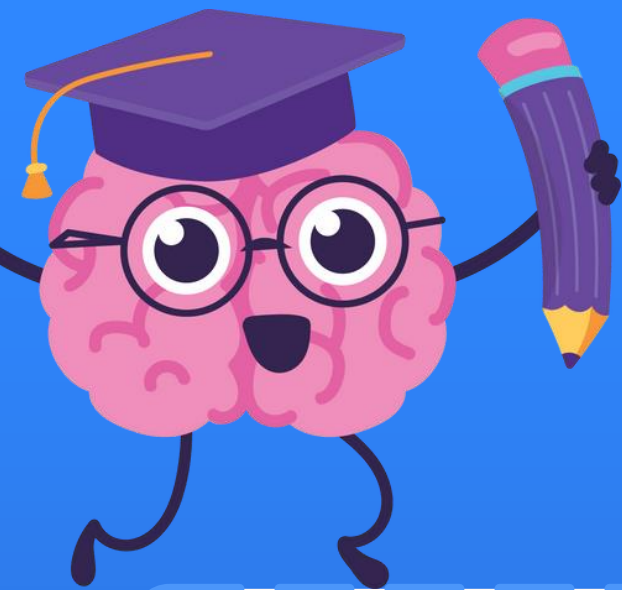
VÍ DỤ

```
#include <iostream>
using namespace std;
class PhanSo
{
private:
    int tu;
    int mau;

public:
    PhanSo(int tu = 0, int mau = 1);
    ~PhanSo();

    // Overload toán tử ++ trước
    PhanSo operator++();
    // Overload toán tử ++ sau
    PhanSo operator++(int);
};
```

```
PhanSo::PhanSo(int tu, int mau)
{
    this->tu = tu;
    this->mau = mau;
}
PhanSo::~~PhanSo() {}
PhanSo PhanSo::operator++()
{
    tu++;
    return *this;
}
PhanSo PhanSo::operator++(int)
{
    PhanSo temp = *this;
    tu++;
    return temp;
}
```



CHƯƠNG VI: DẪN XUẤT VÀ THỪA KẾ

1. Quan hệ giữa các lớp
đối tượng
2. Kế thừa



QUAN HỆ GIỮA CÁC LỚP ĐỐI TƯỢNG



QUAN HỆ GIỮA CÁC LỚP ĐỐI TƯỢNG

Giữa các lớp đối tượng có những loại quan hệ

- Quan hệ một một (1-1)
- Quan hệ một nhiều (1-n)
- Quan hệ nhiều nhiều (n-n)
- Quan hệ đặc biệt hóa, tổng quát hóa

*Ta có: base class: class cha
derived class: class con



QUAN HỆ

1 - 1



QUAN HỆ 1 – 1

- Quan hệ một-một: Một đối tượng thuộc lớp này quan hệ với một đối tượng thuộc lớp kia và một đối tượng thuộc lớp kia có quan hệ duy nhất với một đối tượng thuộc lớp này.
- Kí hiệu:



- Trong hình vẽ trên ta nói: 1 đối tượng thuộc lớp A có quan hệ với 1 đối tượng thuộc lớp B và 1 đối tượng thuộc lớp B có quan hệ duy nhất với 1 đối tượng thuộc lớp A.



QUAN HỆ 1 - 1

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 class Person{
5     private:
6         string name;
7         string address;
8     public:
9         string getName(){
10             return name;
11         }
12         string getAddress(){
13             return address;
14         }
15         void setName(string name){
16             this->name = name;
17         }
18         void setAddress(string address){
19             this->address = address;
20         }
21 };
22
23 class Student : public Person{
24     private:
25         float gpa;
26     public:
27         float getGPA(){
28             return gpa;
29         }
30         void setGPA(float gpa){
31             this->gpa = gpa;
32         }
33 };
```

```
34
35 int main() {
36     Student s;
37     string name, address;
38     float gpa;
39     cin >> name;
40     s.setName(name);
41     cin >> address;
42     s.setAddress(address);
43     cin >> gpa;
44     s.setGPA(gpa);
45     cout << s.getName() << " " << s.getAddress() << " " << s.getGPA();
46     return 0;
47 }
48
49
```

```
toan
vungtau
2.8
toan vungtau 2.8
```

```
...Program finished with exit code 0
Press ENTER to exit console.
```



QUAN HỆ

1 - n



QUAN HỆ 1 - n

- Quan hệ một-nhiều: Một đối tượng thuộc lớp này quan hệ với nhiều đối tượng thuộc lớp kia và một đối tượng lớp kia có quan hệ duy nhất với một đối tượng thuộc lớp này.
- Kí hiệu:



- Trong hình vẽ trên ta nói: 1 đối tượng thuộc lớp A có quan hệ với n đối tượng thuộc lớp B và 1 đối tượng thuộc lớp B có quan hệ duy nhất với 1 đối tượng thuộc lớp A.



QUAN HỆ 1 - n

```
46
47 int main() {
48     int n;
49     cout << "Chon 1(hoc sinh) hay 2(giao vien): ";
50     cin >> n;
51     switch(n){
52     case 1:{
53         Student s;
54         string name, address;
55         float gpa;
56         cin >> name;
57         s.setName(name);
58         cin >> address;
59         s.setAddress(address);
60         cin >> gpa;
61         s.setGPA(gpa);
62         cout << s.getName() << " " << s.getAddress() << " " << s.getGPA();
63         break;
64     }
65     case 2:{
66         Lectuer s;
67         string name, address;
68         float salary;
69         cin >> name;
70         s.setName(name);
71         cin >> address;
72         s.setAddress(address);
73         cin >> salary;
74         s.setSalary(salary);
75         cout << s.getName() << " " << s.getAddress() << " " << s.getSalary();
76         break;
77     }
78     default:
79         cout << "Error! The operator is not correct";
80         break;
81     } return 0;
82 }
```

```
34
35 class Lectuer : public Person{
36     private:
37         float salary;
38     public:
39         float getSalary(){
40             return salary;
41         }
42         void setSalary(float salary){
43             this->salary = salary;
44         }
45     };
46
```

```
Chon 1(hoc sinh) hay 2(giao vien): 2
nhu
vungtau
8000
nhu vungtau 8000k
```

```
...Program finished with exit code 0
Press ENTER to exit console.
```




QUAN HỆ

n – n



QUAN HỆ $n - n$

- Khái niệm: Hai lớp đối tượng được gọi là có quan hệ $n-n$ với nhau khi 1 đối tượng thuộc lớp này có quan hệ với n đối tượng thuộc lớp kia và 1 đối tượng thuộc lớp kia cũng có quan hệ với n đối tượng thuộc lớp này

- Kí hiệu :



- Trong hình vẽ trên ta nói: 1 đối tượng thuộc lớp A có quan hệ với n đối tượng thuộc lớp B và 1 đối tượng thuộc lớp B cũng có quan hệ với n đối tượng thuộc lớp A.



QUAN HỆ n - n

```
23 class Citizens{
24     private:
25         string id;
26         int ages ;
27     public:
28         string getID(){
29             return id;
30         }
31         string getAges(){
32             return ages;
33         }
34         void setID(string ID){
35             this->id = id;
36         }
37         void setAges(string ages){
38             this->ages = ages;
39         }
40     };
41
```

```
class Student : public Person, Citizens{
    private:
        float gpa;
    public:
        Student(string name, string address, float gpa) : Person(name, address), Citizens(id, ages){
            this -> gpa = gpa;
        }

        float getGPA(){
            return gpa;
        }
        void setGPA(float gpa){
            this->gpa = gpa;
        }
};
```



QUAN HỆ n - n

Học sinh

Học

Môn học

Bác sĩ

Chữa bệnh

Bệnh nhân

Khách hàng

Mua

Sản phẩm



QUAN HỆ ĐẶC BIỆT HÓA TỔNG QUÁT HÓA



QUAN HỆ ĐẶC BIỆT HÓA – TỔNG QUÁT HÓA

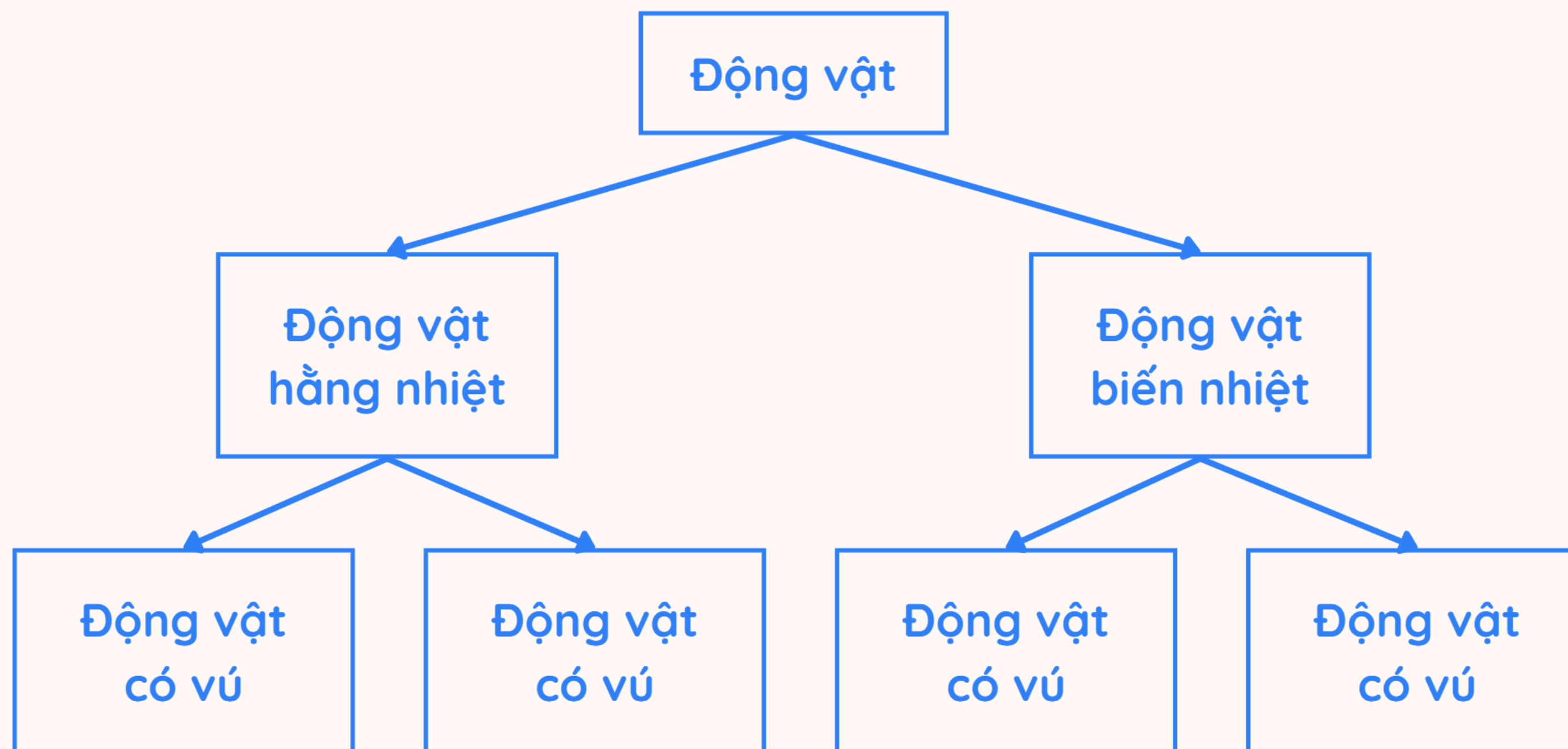
- Khái niệm: Hai lớp đối tượng được gọi là có quan hệ đặc biệt hóa-tổng quát hóa với nhau khi lớp đối tượng này là trường hợp đặc biệt của lớp đối tượng kia và lớp đối tượng kia là trường hợp tổng quát của lớp đối tượng này.
- Kí hiệu:



- Trong hình vẽ trên ta nói: lớp đối tượng B là trường hợp đặc biệt của lớp đối tượng A và lớp đối tượng A là trường hợp tổng quát của lớp đối tượng B.



QUAN HỆ ĐẶC BIỆT HÓA – TỔNG QUÁT HÓA





LỚP CƠ SỞ VÀ LỚP DẪN XUẤT





LỚP CƠ SỞ VÀ LỚP DẪN XUẤT

- Một lớp thừa kế một lớp khác gọi là lớp dẫn xuất (derived class hay sub class).
- Lớp dùng để xây dựng lớp dẫn xuất gọi là lớp cơ sở (super class hay base class).
- Một lớp có thể vừa là lớp cơ sở vừa là lớp dẫn xuất.
- Một lớp có thể dẫn xuất từ nhiều lớp cơ sở và cũng có thể là lớp cơ sở cho nhiều lớp dẫn xuất khác nhau.



LỚP CƠ SỞ VÀ LỚP DẪN XUẤT

Ví dụ: Xây dựng lớp C dẫn xuất từ lớp A và lớp B:

```
class C: public A, public B
```

```
{
```

```
    private:
```

```
        //Khai báo các thuộc tính public:
```

```
        //Các phương thức
```

```
};
```



TÍNH THỬA KẼ



KHÁI NIỆM THỪA KẾ

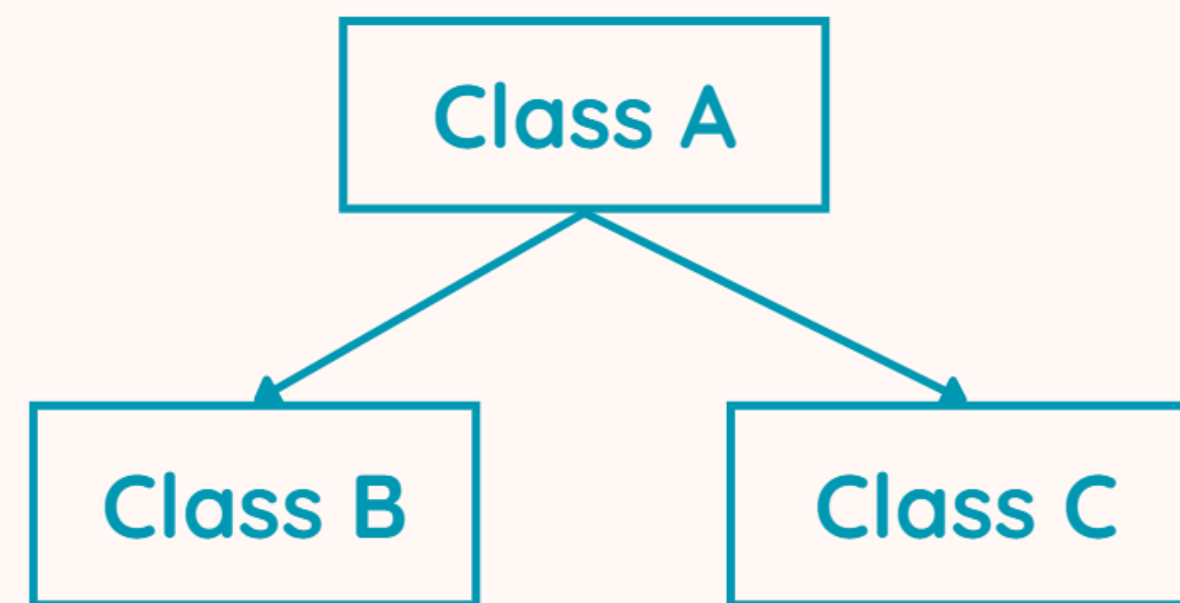
- Tính thừa kế cho phép các lớp được xây dựng trên các lớp đã có.
- Tính thừa kế được dùng để biểu diễn mối quan hệ đặc biệt hóa – tổng quát hóa giữa các lớp. Trong đó, lớp dẫn xuất thừa kế lớp cơ sở.
- Lớp dẫn xuất thừa kế tất cả các thành phần (thuộc tính và phương thức) của lớp cơ sở, kể cả các thành phần mà lớp cơ sở được thừa kế.



KHÁI NIỆM THỪA KẾ

Đặc tính:

- Cho phép định nghĩa lớp mới từ lớp đã có
- Lớp đã có gọi là Lớp Cơ Sở (Base Class) hoặc Lớp Cha (Superclass).
- Lớp mới gọi là Lớp Dẫn Xuất (Derived Class) hoặc Lớp Con (Subclass).





KHAI BẢO THỪA KẾ



KHAI BÁO THỪA KẾ

```
class SuperClass {  
    //Thành phần của lớp cơ sở  
};  
class DerivedClass : public / [private] SuperClass{  
    //Thành phần bổ sung của lớp dẫn xuất  
};
```



KHÁI NIỆM THỪA KẾ

- Ví dụ: Xây dựng Lớp C thừa kế public lớp A và lớp B

```
Student(string name, string address, float gpa) : Person(name, address)
{
    this->gpa = gpa;
};
```

- Lưu ý: nếu không dùng từ khóa nào thì hiểu là private
- Ví dụ: class C: public A, B{...};
- -> Lớp C thừa kế public lớp A
- -> Lớp C thừa kế private lớp B



KHAI BÁO THỪA KẾ

- Việc truy nhập các thành phần của lớp cơ sở được lớp dẫn xuất thừa kế phụ thuộc vào 2 yếu tố:
- Các thành phần đó được khai báo là private hay public hay protected trong lớp cơ sở.
- private: không cho phép truy nhập trong lớp dẫn xuất.
- public: có thể truy nhập tại bất kỳ chỗ nào trong chương trình nên các lớp dẫn xuất có thể truy nhập được.
- protected: chỉ được truy nhập bởi các lớp dẫn xuất trực tiếp. Các thành phần private < các thành phần khai báo là protected có phạm vi truy nhập < các thành phần public



KHAI BẢO THỪA KẾ

- Kiểu dẫn xuất là private (mặc định) hay public hay protected được chỉ định khi định nghĩa lớp dẫn xuất.
- public: các thành phần public và protected của lớp cơ sở sẽ trở thành các thành phần public và protected của lớp dẫn xuất.
- private: các thành phần public và protected của lớp cơ sở sẽ trở thành các thành phần private của lớp dẫn xuất.



CÁC LOẠI THỪA KẾ



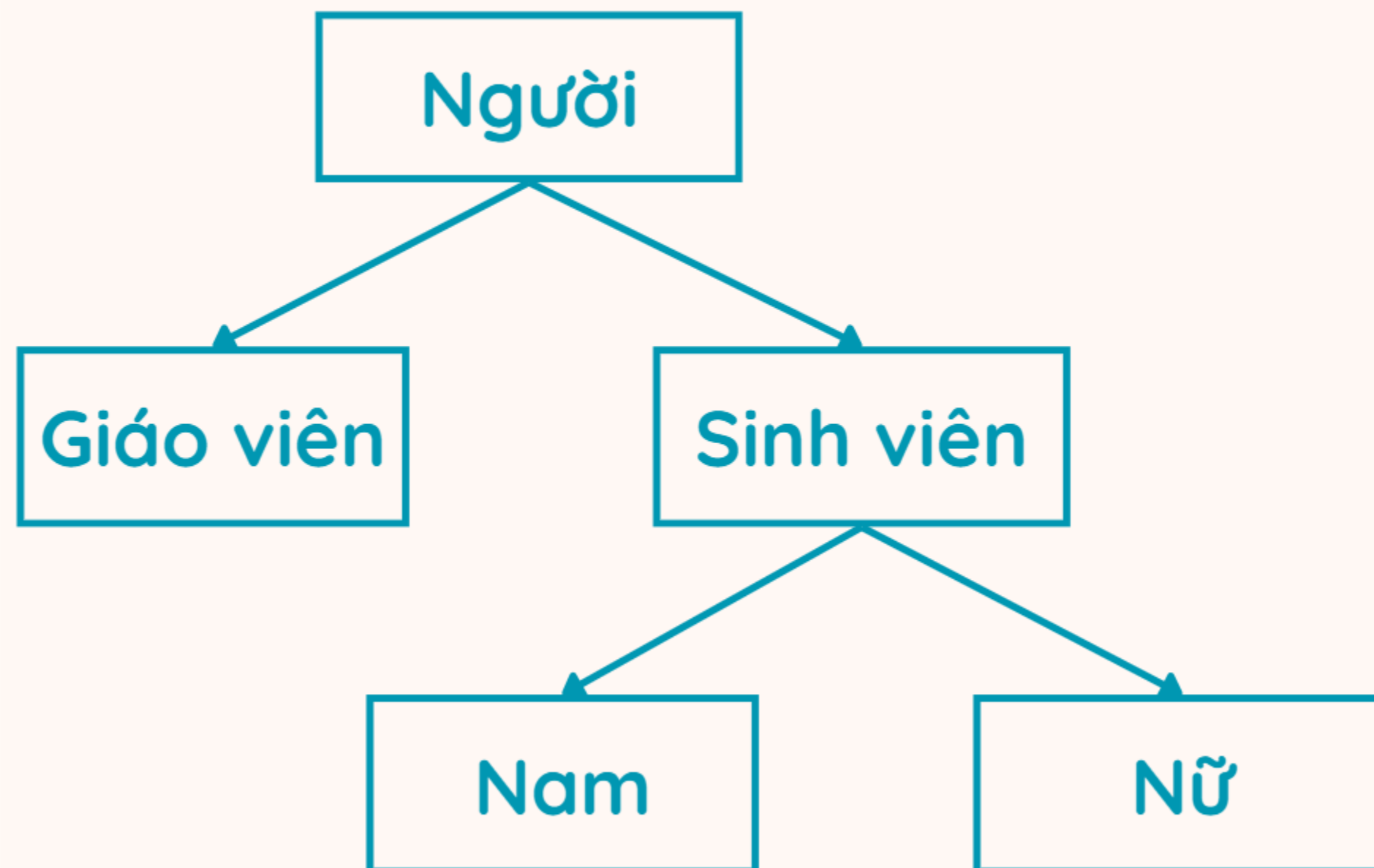
CÁC LOẠI THỪA KẾ

- Đơn thừa kế: lớp được dẫn xuất từ 1 lớp cơ sở. VD: class A : public B {...};
- Đa thừa kế: một lớp có thể được dẫn xuất từ nhiều lớp cơ sở.
- VD: class A : public B, public C {...};
- Ví dụ:

```
Student(string name, string address, float gpa, float ID, int ages) : Person(name, address), Citizen(ID, address)
{
    this->gpa = gpa;
};
```



ĐƠN THỬA KÊ





ĐA THỪA KẾ

- Các đặc điểm của đơn thừa kế vẫn đúng cho trường hợp đa thừa kế.
- Tuy nhiên, trong đa thừa kế có thể có sự nhập nhằng => Khai báo lớp cơ sở ảo.



ĐA THỪA KẾ

```
class A {  
public: int a;  
};  
class B : public A {  
public: int b;  
};  
class C : public A {  
public: int c;  
};  
class D : public B, public C {  
public: int d;  
};
```

ĐA THỪA KẾ

```
int main() {  
    D h;  
    h.a = 1; //? ? ?  
    h.b = 2;  
    h.c = 3;  
    h.d = 4;  
    system("pause");  
    return 0;  
}
```

- Các đặc điểm của đơn thừa kế vẫn đúng cho trường hợp đa thừa kế.
- Tuy nhiên, trong đa thừa kế có thể có sự nhập nhằng => Khai báo lớp cơ sở ảo.



ĐA THỪA KÊ

```
class A {  
public: int a;  
};  
class B : virtual public A {  
public: int b;  
};  
class C : virtual public A {  
public: int c;  
};  
class D : public B, public C {  
public: int d;  
};
```

ĐA THỪA KÊ

```
int main() {  
    D h;  
    h.a = 1; //ok  
    h.b = 2;  
    h.c = 3;  
    h.d = 4;  
    system("pause");  
    return 0;  
}
```

=> Khai báo A là lớp cơ sở kiểu virtual cho cả B và C. Khi đó, hai lớp cơ sở A (A là cơ sở của B và A là cơ sở của C) sẽ kết hợp lại để trở thành 1 lớp cơ sở A duy nhất cho bất kỳ lớp dẫn xuất nào từ B và C.



THỬA KÊ THUỘC TÍNH



THỪA KẾ THUỘC TÍNH

- Gọi A = các thuộc tính của lớp cơ sở
- A được thừa kế trong lớp dẫn xuất:
Tập thuộc tính của lớp dẫn xuất = các thuộc tính mới của lớp dẫn xuất + A
- Tuy nhiên trong các phương thức của lớp dẫn xuất không cho phép truy nhập đến các thuộc tính private của lớp cơ sở.



THỬA KÊ PHƯƠNG THỨC



THỪA KẾ PHƯƠNG THỨC

Lớp dẫn xuất không thừa kế các HÀM TẠO, HÀM HỦY, HÀM TOÁN TỬ GÁN của các lớp cơ sở. - Các phương thức public khác của lớp cơ sở được thừa kế trong lớp dẫn xuất.



THỪA KẾ PHƯƠNG THỨC

Tuy nhiên, các phương thức của lớp cơ sở có thể được cài đặt lại trong lớp dẫn xuất trong một số trường hợp sau:

- TH1: Thao tác ở lớp dẫn xuất khác thao tác ở lớp cơ sở. Thông thường là các thao tác nhập, xuất.
- TH2: Giải thuật ở lớp dẫn xuất đơn giản hơn (ví dụ như tô màu đa giác, tính diện tích...).
- TH3: Thao tác không có ý nghĩa trong lớp dẫn xuất (ví dụ hàm quay 1 hình tròn (lớp hình tròn thừa kế lớp elip)).
- TH4: Các phương thức có thể vi phạm ràng buộc dữ liệu của lớp dẫn xuất (ví dụ cài đặt lại toán tử gán để đảm bảo ràng buộc là mọi đối tượng thuộc lớp Số ảo phải có phần thực bằng 0).



**LỚP CƠ SỞ LÀ
THÀNH PHẦN
CỦA LỚP DẪN
XUẤT**



LỚP CƠ SỞ LÀ THÀNH PHẦN CỦA LỚP DẪN XUẤT

- Có thể thay thừa kế bằng cách khai báo lớp cơ sở là thành phần của lớp dẫn xuất.
- Khi đó lớp dẫn xuất là lớp bao => xây dựng hàm tạo của lớp bao sẽ sử dụng các hàm tạo của các lớp thành phần tương ứng để khởi gán cho các thuộc tính là đối tượng của lớp bao.



LỚP CƠ SỞ LÀ THÀNH PHẦN CỦA LỚP DẪN XUẤT

```
class Student : public Person, {  
    private:  
        float gpa;  
    public:  
        Student : Person{  
            this->gpa = gpa;  
        }  
  
        float getGPA() {  
            return gpa;  
        }  
        void setGPA(float gpa) {  
            this->gpa = gpa;  
        }  
};
```

```
class Student {  
    private:  
        float gpa;  
    public:  
        Person(name, address);  
        this->gpa = gpa  
        float getGPA() {  
            return gpa;  
        }  
        void setGPA(float gpa) {  
            this->gpa = gpa;  
        }  
};
```




PHẠM VI TRUY XUẤT



PHẠM VI TRUY XUẤT

- Mặc dù lớp dẫn xuất được thừa kế tất cả các thành phần của lớp cơ sở (thuộc tính và phương thức), nhưng trong lớp dẫn xuất không thể truy nhập đến tất cả các thành phần này.
- Thường thì các thuộc tính của lớp cơ sở sẽ được truy nhập thông qua các phương thức public mà lớp cơ sở cung cấp.



PHẠM VI TRUY XUẤT

Việc truy nhập đến các thành phần của lớp cơ sở phụ thuộc vào 2 yếu tố:

- Các thành phần đó được khai báo là private (mặc định) hay public hay protected trong lớp cơ sở. => Truy xuất theo chiều dọc.
- Kiểu dẫn xuất là private (mặc định) hay public hay protected được chỉ định khi định nghĩa lớp dẫn xuất. => Truy xuất theo chiều ngang.



CÁC HÀM TRONG THỬA KẼ





HÀM TẠO, HÀM HỦY VÀ HÀM TOÁN TỬ GÁN

Lớp dẫn xuất không thừa kế các HÀM TẠO, HÀM HỦY, HÀM TOÁN
TỬ GÁN của các lớp cơ sở.



HÀM TẠO CỦA LỚP DẪN XUẤT

- Vì trong lớp dẫn xuất không truy nhập được các thuộc tính là thành phần private của lớp cơ sở nên để gán giá trị cho các thuộc tính này phải sử dụng hàm tạo của lớp cơ sở (hàm tạo là thành phần public).

=> Lớp cơ sở phải cung cấp hàm tạo có đối để gán giá trị cho các thuộc tính là thành phần private của nó. Cách dùng:

Tên_lớp_cơ_sở (danh sách giá trị)



PHẠM VI TRUY XUẤT

- Ngoài ra, nếu lớp dẫn xuất có thành phần kiểu lớp (đối tượng thành phần) thì phải dùng hàm tạo của lớp tương ứng để gán giá trị cho các thuộc tính của đối tượng này vì trong lớp dẫn xuất không truy nhập được các thuộc tính này.

=> Lớp tương ứng phải cung cấp hàm tạo có đối để gán giá trị cho các thuộc tính là thành phần private của nó.

Cách dùng:

Tên_đối_tượng_thành_phần(danh sách giá trị)



HÀM TOÁN TỬ GÁN CỦA LỚP DẪN XUẤT

Khi lớp dẫn xuất có thuộc tính là con trỏ (kể cả thuộc tính thừa kế từ các lớp cơ sở) thì phải xây dựng hàm toán tử gán cho lớp dẫn xuất. Cách xây dựng:

- Xây dựng hàm toán tử gán cho lớp cơ sở;
- Xây dựng hàm trả về địa chỉ của đối tượng ẩn của lớp cơ sở theo mẫu:

```
Tên_lớp * get_DTA(){ return this; }
```

- Xây dựng hàm toán tử gán cho lớp dẫn xuất.



HÀM TẠO SAO CHÉP CỦA LỚP DẪN XUẤT

Khi lớp dẫn xuất có thuộc tính là con trỏ (kể cả thuộc tính thừa kế từ các lớp cơ sở) thì phải xây dựng hàm tạo sao chép cho lớp dẫn xuất.

Cách xây dựng:

- Xây dựng hàm toán tử gán cho lớp dẫn xuất;
- Xây dựng hàm tạo sao chép cho lớp dẫn xuất.



HÀM HỦY CỦA LỚP DẪN XUẤT

- Khi xây dựng hàm hủy của lớp dẫn xuất chỉ cần quan tâm đến các thuộc tính khai báo thêm trong lớp dẫn xuất.
- Trong đó, nếu thuộc tính có kiểu lớp (đối tượng thành phần) thì hàm hủy của lớp đó sẽ được tự động gọi.
- Như vậy ta không cần để ý đến các đối tượng thành phần và các thuộc tính thừa kế từ các lớp cơ sở trong hàm hủy của lớp dẫn xuất.



HÀM HỦY CỦA LỚP DẪN XUẤT

- Khi xây dựng hàm hủy của lớp dẫn xuất chỉ cần quan tâm đến các thuộc tính khai báo thêm trong lớp dẫn xuất.
- Trong đó, nếu thuộc tính có kiểu lớp (đối tượng thành phần) thì hàm hủy của lớp đó sẽ được tự động gọi.
- Như vậy ta không cần để ý đến các đối tượng thành phần và các thuộc tính thừa kế từ các lớp cơ sở trong hàm hủy của lớp dẫn xuất.



HÀM HỦY CỦA LỚP DẪN XUẤT

```
class SinhVien : public Nguoi {  
    char* MaSo;  
public:  
    SinhVien(char* ht, char* ms, int ns) : Nguoi(ht, ns) {  
        MaSo = _strdup(ms);  
    } /*Hàm tạo sao chép của lớp SinhVien*/  
    SinhVien(const SinhVien& s) : Nguoi(s)  
    { /*Hàm tạo sao chép của lớp Nguoi*/  
        MaSo = _strdup(s.MaSo);  
    }  
    ~SinhVien() { delete[] MaSo; }  
};
```

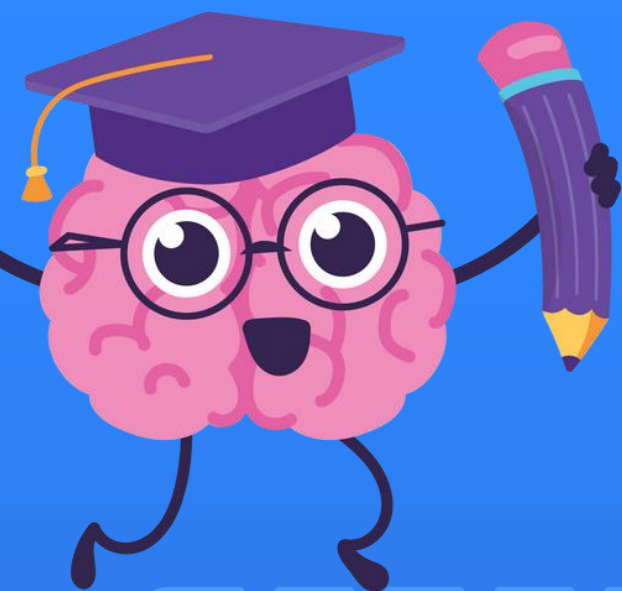



LỢI ÍCH



LỢI ÍCH

- Xây dựng lớp mới từ lớp đã có.
- Chia sẻ mã chương trình chung \Rightarrow Dễ sửa chữa, nâng cấp.
- Cơ chế chuyển kiểu tự động trong C++.
- Dễ dàng bổ sung hoặc định nghĩa lại các thành phần.



BAN HỌC TẬP
HỆ THỐNG THÔNG TIN

150



CHƯƠNG VII:

TÍNH

ĐA

THỪA KẾ

1. Quan hệ giữa các lớp

đối tượng

2. Kế thừa



PHƯƠNG THỨC TĨNH





PHƯƠNG THỨC TĨNH

Là phương thức không phụ thuộc vào trạng thái của đối tượng, mà chỉ phụ thuộc vào lớp mà nó được khai báo.

```
class A {  
public:  
    void Xuat() { cout << "\n Lop A"; }  
};  
class B : public A {  
public:  
    void Xuat() { cout << "\n Lop B"; }  
};  
class C : public A {  
public:  
    void Xuat() { cout << "\n Lop C"; }  
};  
class D : public C {  
public:  
    void Xuat() { cout << "\n Lop D"; }  
};
```

```
void HienThi(A* p) { //Hàm toàn cục  
    p->Xuat();  
}  
void main() {  
    A a;  
    B b;  
    C c;  
    D d;  
    HienThi(&a); //Gọi tới A::Xuat()  
    HienThi(&b); //Gọi tới A::Xuat()  
    HienThi(&c); //Gọi tới A::Xuat()  
    HienThi(&d); //Gọi tới A::Xuat()  
}
```



PHƯƠNG THỨC ẢO





PHƯƠNG THỨC ẢO

- Là phương thức cho phép các lớp con định nghĩa lại hành vi của phương thức được khai báo trong lớp cha, tạo ra sự đa dạng trong hành vi của cùng một phương thức.

PHƯƠNG THỨC ẢO

- Phương thức ảo chỉ khác phương thức tĩnh khi được gọi từ một con trỏ.
- Phương thức được gọi phụ thuộc vào đối tượng cụ thể mà con trỏ đang trỏ tới: con trỏ đang trỏ tới đối tượng của lớp nào thì phương thức của lớp đó sẽ được gọi.



ĐỊNH NGHĨA PHƯƠNG THỨC ẢO





ĐỊNH NGHĨA PHƯƠNG THỨC ẢO

Cách 1: Thêm từ khóa **virtual** vào dòng tiêu đề của phương thức trùng tên **bên trong định nghĩa** của lớp cơ sở.

Cách 2: Thêm từ khóa **virtual** vào dòng tiêu đề của phương thức trùng tên **bên trong định nghĩa** của lớp cơ sở và tất cả lớp dẫn xuất.

Lưu ý: Nếu phương thức được cài đặt **bên ngoài định nghĩa** lớp thì từ khóa **virtual** chỉ xuất hiện ở **prototype** của phương thức (đặt **bên trong định nghĩa** lớp).



CÁCH 1

```
class A {  
public:  
    virtual void Xuat() { cout << "\n Lop A"; }  
};  
class B : public A {  
public:  
    void Xuat() { cout << "\n Lop B"; }  
};  
class C : public B {  
public:  
    void Xuat() { cout << "\n Lop C"; }  
};  
class D : public C {  
public:  
    void Xuat() { cout << "\n Lop D"; }  
};
```

CÁCH 2

```
class A {  
public:  
    virtual void Xuat() { cout << "\n Lop A"; }  
};  
class B : public A {  
public:  
    virtual void Xuat() { cout << "\n Lop B"; }  
};  
class C : public B {  
public:  
    virtual void Xuat() { cout << "\n Lop C"; }  
};  
class D : public C {  
public:  
    virtual void Xuat() { cout << "\n Lop D"; }  
};
```



CÁCH GỌI PHƯƠNG THỨC ẢO





CÁCH GỌI PHƯƠNG THỨC ẢO

```
class A {  
public:  
    virtual void Xuat() { cout << "\n Lop A"; }  
};  
class B : public A {  
public:  
    void Xuat() { cout << "\n Lop B"; }  
};  
class C : public B {  
public:  
    void Xuat() { cout << "\n Lop C"; }  
};  
class D : public C {  
public:  
    void Xuat() { cout << "\n Lop D"; }  
};
```

```
void main() {  
    A* p; //p là con trỏ kiểu A  
    A a;  
    B b;  
    C c;  
    D d;  
    p = &a;  
    p->Xuat(); //Gọi tới A::Xuat()  
    p = &b;  
    p->Xuat(); //Gọi tới B::Xuat()  
    p = &c;  
    p->Xuat(); //Gọi tới C::Xuat()  
    p = &d;  
    p->Xuat(); //Gọi tới D::Xuat()  
}
```




LIÊN KẾT TĨNH

- Lời gọi tới phương thức tĩnh (xuất phát từ con trỏ) luôn luôn liên kết với một phương thức cố định.
- Sự liên kết này được xác định trong quá trình biên dịch chương trình.

LIÊN KẾT ĐỘNG

- Lời gọi tới phương thức ảo (xuất phát từ con trỏ) không liên kết với một phương thức cố định mà tùy thuộc vào nội dung con trỏ.
- Phương thức mà nó liên kết chưa xác định trong giai đoạn dịch chương trình.
- Phương thức được liên kết thay đổi mỗi khi có sự thay đổi nội dung con trỏ trong quá trình chạy chương trình.



CHÚ Ý

Phương thức ảo phải được định nghĩa thống nhất từ lớp cơ sở cho đến các lớp dẫn xuất:

- Trùng kiểu trả về;
- Trùng tên phương thức;
- Trùng danh sách đối.



ỨNG DỤNG CỦA PHƯƠNG THỨC ẢO





ỨNG DỤNG CỦA PHƯƠNG THỨC ẢO

- Có thể dùng phương thức ảo để giải quyết bài toán quản lý một danh sách các đối tượng có kiểu khác nhau. Khi đó, tùy thuộc vào kiểu của từng đối tượng mà phương thức tương ứng sẽ được gọi.
- Phương thức ảo làm cho việc bổ sung nâng cấp chương trình dễ dàng hơn: thêm các lớp dẫn xuất và cài đặt lại phương thức ảo của lớp cơ sở mà không cần phải sửa đổi trên lớp cơ sở (đảm bảo tính đóng gói).



TÍNH ĐA HÌNH





TÍNH ĐA HÌNH

- Tính đa hình xuất hiện khi có sự thừa kế giữa các lớp.
 - Tính đa hình muốn nói đến tình huống cùng 1 câu lệnh (vd: `p->Xuat();`) nhưng trong hai trường hợp khác nhau (vd: gọi đến `Diem::Xuat();` hoặc `Diemmau::Xuat();`) có thể xuất ra hai kết quả khác nhau.
- => Tính đa hình cho phép xử lý nhiều đối tượng khác nhau, nhiều công việc khác nhau theo cùng 1 cách thức.
- Các phương thức thể hiện tính đa hình phải được định nghĩa là các phương thức ảo trong lớp cơ sở. Các phương thức này sẽ được các lớp dẫn xuất cài đặt lại theo cách riêng của mình.



HÀM THUẦN ẢO VÀ LỚP TRỪU TƯỢNG

- Hàm thuần ảo là hàm không có phần định nghĩa.
- Lớp có ít nhất một hàm thuần ảo gọi là Lớp trừu tượng (Abstract class).

Ví dụ:

```
class A { //Lớp trừu tượng
public:
    virtual void f1() = 0; //Hàm ảo thuần túy
    virtual void f2() = 0; //Hàm ảo thuần túy
};
```



HÀM THUẦN ẢO VÀ LỚP TRỪU TƯỢNG

Lưu ý:

- Không thể sử dụng lớp trừu tượng để khai báo (tạo ra) đối tượng vì nó chỉ được dùng để mô tả lớp các đối tượng chung nhất.
- Được phép khai báo con trỏ kiểu lớp trừu tượng.
- Một hàm ảo thuần túy khai báo trong một lớp trừu tượng cơ sở phải được định nghĩa lại trong lớp dẫn xuất hoặc nếu không thì phải được tiếp tục khai báo ảo trong lớp dẫn xuất.



BAN HỌC TẬP
HỆ THỐNG THÔNG TIN

